TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

DESARROLLO DE UN PROTOTIPO DE ENTORNOS VIRTUALES PARA FINES DIDÁCTICOS EN EMPRESAS

AUTOR: Rafael Román Aguilar

Puerto Real, 19 de marzo de 2021

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

DESARROLLO DE UN PROTOTIPO DE ENTORNOS VIRTUALES PARA FINES DIDÁCTICOS EN EMPRESAS

DIRECTOR: Guillermo Bárcena González
CODIRECTOR: Juan Carlos de la Torre Macías
AUTOR: Rafael Román Aguilar

Puerto Real, 19 de marzo de 2021

Agradecimientos

A mis padres, por creer siempre en mí y apoyarme en todo lo que me llena.

A Pablo, por ser como es y gustarle tanto este proyecto.

A mi familia, por darme ánimos y confiar ciegamente en mí.

A Paula y Víctor, por aguantarme y escucharme todo este tiempo, siempre hacéis que me olvide de lo malo.

A todos los amigos que he hecho en la carrera, gracias a vosotros estos cuatro años han supuesto un camino de flores, y en especial a Borja, con quién compartía risas todas las mañanas antes de clase.

A los integrantes del SIC, por echarme un cable siempre que lo he necesitado y facilitarme el material que requería para este proyecto, y en especial a Guillermo y Juan Carlos, por guiarme durante la realización del mismo.

Resumen Debido al avance de las tecnologías, el ser humano es capaz de interactuar con un sistema virtual de muchas maneras distintas, ya sea con nuestras propias manos como es el caso del dispositivo *Leap Motion*, con nuestra voz (*Alexa, Echo Dot de Amazon*), étc. Incluso podemos visualizar un entorno como si nos encontráramos en él con el casco de realidad virtual *Oculus Rift*.

Ante esto, muchas empresas han aprovechado la oportunidad para integrar sus productos y crear aplicaciones orientadas a enseñanza, simulaciones o prácticas utilizando las tecnologías de manera simultánea y conjunta.

Este trabajo pretende profundizar en este tema ya que nuestra intención es crear un prototipo de entorno que integre a todas estas tecnologías y que pueda ser utilizado en un futuro para tareas didácticas en empresas.

Palabras clave Realidad virtual, control por voz, detección de manos, inteligencia artificial, interacción, entornos

Abstract Due to the growth of technology, human being is capable of interacting with a virtual system in many differents ways, either with our own hands as in the case of the device *Leap Motion*, with our voice (Alexa, Amazon's Echo Dot), etc. We can even visualize an environment as if we were there with the virtual reality headset *Oculus Rift*.

Therefore, a lot of companies have taken the opportunity to integrate their products and create applications oriented to education, simulations or practice using the technologies simultaneously and together.

This work will deepen in this topic because our intention is to create an environment prototype that will integrate all these technologies and that it can be used in the future for didactic tasks in companies.

Keywords Virtual reality, voice control, hands detection, artificial intelligence, interaction, environments

Índice

Índice de figuras

Siglas

VR Virtual Reality (Realidad Virtual)
AWS Amazon Web Services
SIC Laboratorio de Sistemas Inteligentes y de Computación
API Application Programming Interfaces (Interfaz de Programación de Aplicaciones)
AVS Alexa Voice Service (Servicio de Voz de Alexa)
NLU Natural Language Understanding (Comprensión Natural del Lenguaje)
ASR Automatic Speech Recognition (Reconocimiento Automático del Habla)
ASK Alexa Skill Kit
SQS Simple Queue Service (Servicio de Cola de Mensajes)
EC2 Elastic Compute Cloud (Ordenador Elástico en la Nube)
AMI Amazon Image Machine (Imagen de la Máquina Virtual de Amazon)
IA Inteligencia Artificial
HTTP Hypertext Transfer Protocol (Protocolo de Transferencia de Hipertexto)
URL Uniform Resource Locator (Localizador de Recursos Uniforme)
SDK Software Development Kit (Kit de Desarrollo)

1. Introducción

A la hora de probar una aplicación virtual, el pensamiento inicial que está impuesto en la mayoría de personas es que las **interacciones posibles** serán *mover* y *mirar* diversos objetos mientras nos desplazamos por el entorno y que para ello tendremos que manejar un mando o controlador. Sin embargo, en los últimos tiempos las aplicaciones virtuales han avanzado mucho más que eso.

Por un lado, las interacciones que podemos realizar con el entorno son infinitas (a gusto del creador) ya que según una serie de acciones podríamos cambiar el tamaño, forma o color del entorno que nos rodea o incluso de los objetos de este. La prueba de ello se encuentra en videojuegos como *Superliminal*[1] donde el tamaño de los objetos y el entorno se verán modificados según la perspectiva que tomemos de ellos o la aplicación *NonEuclidean*[2] creada en Unity, donde se simula un mundo en el que las distancias no son euclídeas.

Por otro lado, con el reconocimiento de patrones e imágenes hemos podido lograr tecnologías que identifiquen en tiempo real nuestra voz, las posiciones de nuestras manos, cabeza, étc. Con esto, tecnologías como *Leap Motion* o *Alexa* nos permiten la ausencia de mandos para manejar una aplicación virtual. Ejemplo de esto es *Blocks*[3], un videojuego gratuito de los desarrolladores de *Leap Motion* donde con un casco VR podemos crear distintos cubos y modificar su forma o tamaño con nuestras propias manos.

1.1. Motivación

En el momento en que aparecieron en escena cascos de realidad virtual, tecnologías como Unity comenzaron a adaptar sus recursos para integrar este tipo de dispositivos de manera fácil y cómoda. Tras esto, muchas otras empresas comenzaron a introducir sus productos en Unity. Esto, sumado a la facilidad que posee para crear aplicaciones o sistemas, lo ha convertido en una especie de enlace para unir diversos dispositivos con ciertas funciones.

Por ello, podemos encontrar un sinfín de estas aplicaciones que mezclan distintas tecnologías. Sin embargo, es poco común encontrarlas orientadas a un ámbito didáctico o de enseñanza que pueda ser usado en un entorno empresarial. Es por ello que este trabajo se enfoca en producir un **prototipo** uniendo las tecnologías mencionadas anteriormente, que posea diversas **formas de interacción** y permita **simulaciones verosímiles** que puedan ser usadas en **labores didácticas** dentro del ámbito empresarial.

1.2. Finalidad

La finalidad de este trabajo es obtener un prototipo sobre el que se construirían futuros proyectos sobre todo orientados al ámbito laboral o de aprendizaje, aunque podría extrapolarse al ámbito de los videojuegos o el entretenimiento.

Es decir, el pensamiento inicial de este proyecto fue su uso en enseñanza o simulaciones del ámbito laboral y empresarial, como podría ser aprender a manejar la cabina de controles de un barco, las acciones necesarias para repostar un avión o incluso la forma de apretar los tornillos de un coche. Además esto nos permitiría abaratar costes de materiales e incluso asegurar la integridad de los trabajadores ya que estas simulaciones supondrían una instrucción previa al proceso real.

Es en este momento donde todas las tecnologías anteriormente mencionadas cobran sentido, ya que con el casco VR *Oculus Rift* visualizamos el entorno de manera natural, con el dispositivo *Leap Motion* manejamos los controles, objetos o máquinas y con *Alexa* cambiamos la simulación o el entorno (por ejemplo pasar a otro tipo de avión). Por último, Unity nos sirve como forma de aunar todo lo anterior y determinar cómo será nuestro sistema.

1.3. Objetivos

Este proyecto consta de varios objetivos:

- **Integrar las distintas tecnologías de manera natural.** Esto quiere decir que ninguna sea utilizada de manera forzada u obligada, sino que para obtener una experiencia completa del entorno haya que recurrir a cada una de ellas en algún momento.
- **Crear una serie de elementos propios del entorno.** Estos nos permitirán demostrar el potencial que posee el proyecto y será el resultado final de las interacciones que realicemos.
- **Definir las interacciones posibles entre el usuario y el entorno.** Decidir las capacidades que tiene el usuario para modificar el entorno y qué tecnología será la más eficiente en cada caso.
- **Integrar Inteligencia Artificial.** Adaptar al entorno alguna forma de Inteligencia Artificial de manera que podamos apoyar o ayudar al usuario a la hora de realizar alguna actividad dentro de este.

Cumpliendo estos objetivos tendremos listo un entorno virtual con un gran potencial.

1.4. Planificación

El desarrollo de este proyecto se divide en etapas como se puede apreciar en la fig.1, constando de las siguientes:

- **Etapa 1: Investigación y aprendizaje de las tecnologías utilizadas**. Debido a que el proyecto se inició como parte del programa de *Alumno Colaborador*, las tecnologías utilizadas en este eran desconocidas en un primer momento, por lo que debíamos familiarizarnos con ellas y aprender sus capacidades.

- **Etapa 2: Creación del entorno virtual**. En esta etapa, una vez se hubo comprendido las tecnologías, se usó Unity para crear un entorno que aunara *Oculus Rift*, *Leap Motion* y diversos objetos (piezas del tetris) creados por nosotros que nos permitían comprobar el potencial de la aplicación.

- **Etapa 3: Implementación de la comunicación entre Alexa y Unity**. Esta etapa es la que ocupa más tiempo ya que tuvimos que investigar varias tecnologías y decidir una manera de construir un puente de comunicación entre Alexa y Unity. En esta etapa se trata AWS Lambda, CloudAMQP y la manera de construir *skills* de Alexa.

- **Etapa 4: Creación del tablero**. Durante la realización de esta etapa se originó un tablero para las simulaciones que fuera eficiente, con buena estructura de datos y una gran definición de su comportamiento o mecánicas, de forma que interactuar con él resultara natural.

- **Etapa 5: Simulaciones e Inteligencia Artificial**. En esta etapa, usando el tablero de la anterior, construimos una serie de reglas y elementos con los que formamos simulaciones. Además, incorporamos elementos de Inteligencia Artificial a las mismas.

- **Etapa 6: Generación de Objetos**. Durante esta etapa las tareas principales fueron la búsqueda de una colección de objetos 3D, aprender a usar su API y la incorporación de un repositorio que nos permitiera en tiempo de ejecución añadir estos modelos 3D al entorno.

Durante los años 2018/2019 y 2019/2020 fui alumno colaborador en el laboratorio de Sistemas Inteligentes y de Computación (SIC), sirviendo este último para realizar el trabajo que estamos tratando. Es por ello que podemos ver en el *Digrama de Gantt* de la fig.1 que este proyecto se ha realizado a lo largo de un año, ya que comenzando como *alumno colaborador* empezamos con una idea inicial (la base de este trabajo) y fuimos añadiendo características y mejoras a lo largo del tiempo.

Figura 1: Diagrama de Gantt con las etapas de desarrollo

1.5. Estructura

Este proyecto se divide en varios capítulos que recogen los aspectos más importantes relacionados con el desarrollo del susodicho entorno virtual:

- **Capítulo 2: Estado del arte.** En este capítulo discutiremos las innovaciones que se están produciendo en las distintas tecnologías y proyectos similares que de alguna forma se relacionan con el nuestro.
- **Capítulo 3: Fundamentos Teóricos.** En este capítulo recogeremos los fundamentos teóricos que poseen las tecnologías utilizadas. Por tanto, explicaremos en profundidad en qué consiste cada una de ellas y cuál es su potencial.
- **Capítulo 4: Desarrollo.** En este capítulo trataremos el desarrollo del proyecto siguiendo un orden cronológico y explicando cada uno de los conceptos implementados.
- **Capítulo 5: Conclusiones.** Analizaremos el prototipo final y discutiremos los resultados y trabajo futuro.
- **Capítulo 6: Anexo.** En este apartado incluiremos un manual de usuario para recoger los requisitos y formas de uso de nuestra aplicación.

1.6. Metodología

En este proyecto, la metodología ha sido un concepto clave en cada momento ya que al poseer varios objetivos que tratan conceptos dispares no debemos olvidar cuál es la finalidad y qué añade al prototipo definitivo cada uno de ellos.

Por ejemplo, cuando nos encontramos en la etapa de integración de tecnologías tenemos que tener claro que iremos introduciendo cada tecnología de una en una y probando la cohesión entre ellas. Es por ello que cada apartado consiste en la investigación del concepto a implementar y un debate sobre cómo llevar a cabo esta implementación.

Nuestra metodología consiste en que el concepto a implementar **funcione**; y una vez lo haga, **mejorarlo** o incluso **reforzarlo** para evitar problemas futuros. Esto, para el desarrollo del entorno ha sido la manera de enfocar los objetivos y resultados. Gracias a esta metodología hemos podido hacer nuestro prototipo de manera **escalonada** o "por capas" ya que las funcionalidades se han ido introduciendo cuando estaban capacitadas para el uso, es decir, **finalizadas**.

En la figura 2 podemos observar un esquema que muestra la metodología que hemos explicado anteriormente. En esta se muestra el flujo que sigue un objetivo antes de ser considerado como *finalizado* en nuestro proyecto.

Por último, en la figura 3 encontramos el **mapa conceptual del proyecto**, en este se muestran las características que añadimos a nuestro trabajo (azul), las necesidades que encontramos con cada una (blanco) y las soluciones que propusimos (verde). Este **mapa conceptual** nos permite obtener una vista amplia y general de las capacidades que posee nuestra aplicación.

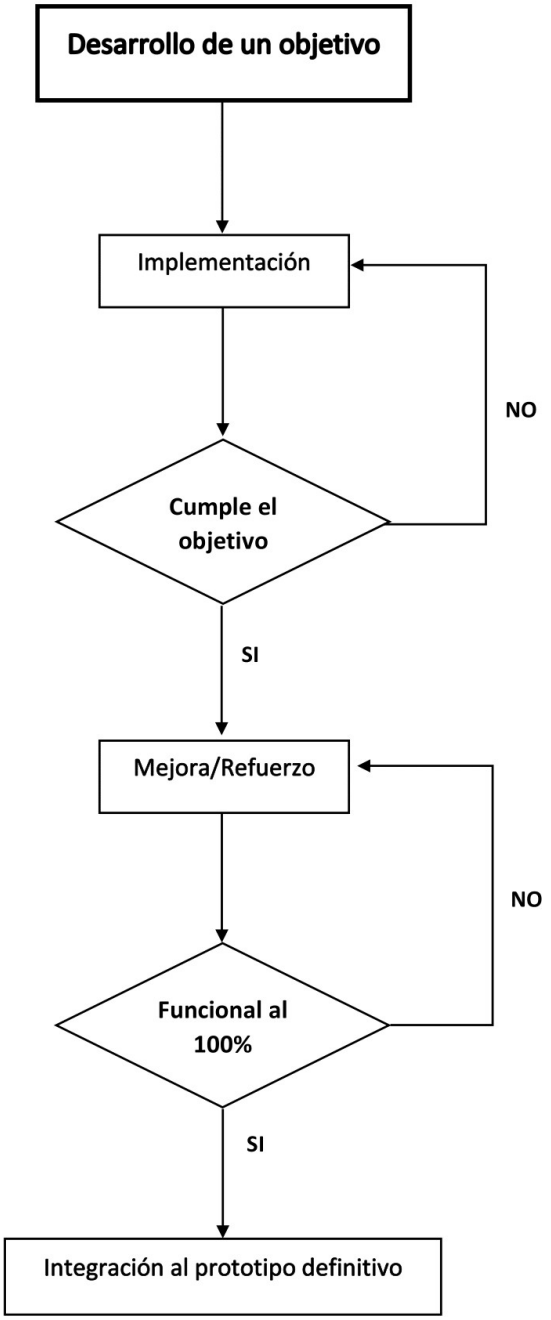

Figura 2: Esquema que ilustra la metodología del proyecto

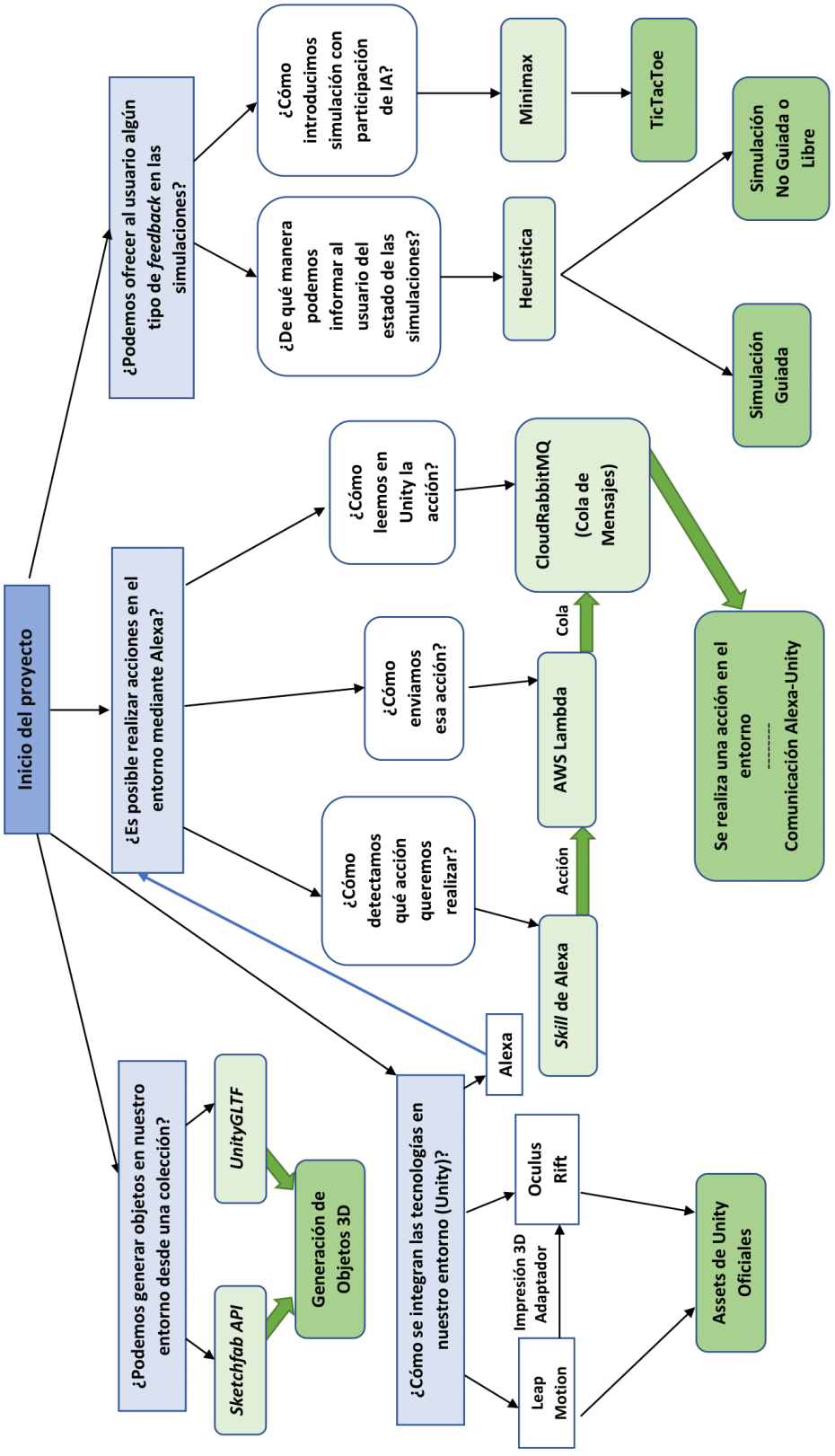

Figura 3: Mapa conceptual del proyecto

2. Estado del Arte

En esta sección discutiremos los avances hasta la fecha que se han realizado en los campos relacionados con este proyecto. Como en el mismo abarcamos diferentes tecnologías, nuestra intención es dar un trasfondo de cada una de ellas de forma que pueda entenderse mejor los logros realizados en este trabajo.

Además, otro punto importante a tener en cuenta en esta sección es la capacidad de evolución que tienen estas tecnologías, de forma que muchas pueden ver mejoras o cambios de diseño en tan sólo algunos meses. Es por ello que presentaremos algunos de estos avances y así podremos comparar con el estado en el que se encontraban al inicio del proyecto.

2.1. Alexa

Desde su lanzamiento en *2014*, Amazon ha permitido que los usuarios desarrollen sus propias *skills*. Desde el punto de vista de los desarrolladores de skills, Alexa no ha cambiado demasiado desde entonces. Esta herramienta de desarrollo nos ha permitido crear la interacción que existe a través de Alexa entre el **entorno** y el **usuario**.

Aunque la herramienta de desarrollo de skills no ha presentado muchos cambios, sí lo ha hecho el hardware que aloja a Alexa ya que el primer dispositivo era el *Echo Dot*, que consistía en un altavoz con una serie de botones. Con el paso del tiempo, este ha ido presentando rediseños y mejoras (actualmente se encuentra en la 4ª generación) e incluso se han presentado nuevos dispositivos como puede ser el *Echo Show* (véase fig.4) que posee pantalla o el *Echo Auto* que está preparado para incorporarlo a nuestro coche.

2.2. Nuevos cascos de Realidad Virtual

Para la realización de este proyecto, el laboratorio SIC contaba con el casco VR **Oculus Rift CV1** que fue lanzado en el 2016 y adquirido en 2018; por tanto, desde el inicio del trabajo el único casco VR utilizado ha sido este. Durante el desarrollo, en Octubre de 2020, Oculus decidió sacar un modelo más actualizado: **Oculus Quest 2**.

La principal mejora de **Oculus Quest 2** se encuentra en las diferentes cámaras y sensores que posee el casco integrado (como podemos ver en la fig.5) gracias a los cuáles ahora no es necesario colocar sensores estáticos frente a las gafas[4] (como ocurre con las CV1), con los respectivos problemas y limitaciones que esto conllevaba. Estas cámaras permiten tener un rastreo de nuestra posición, de los mandos (*Oculus Touch*) y gracias a una actualización lanzada en 2019, también nos permite el rastreo de nuestras manos.

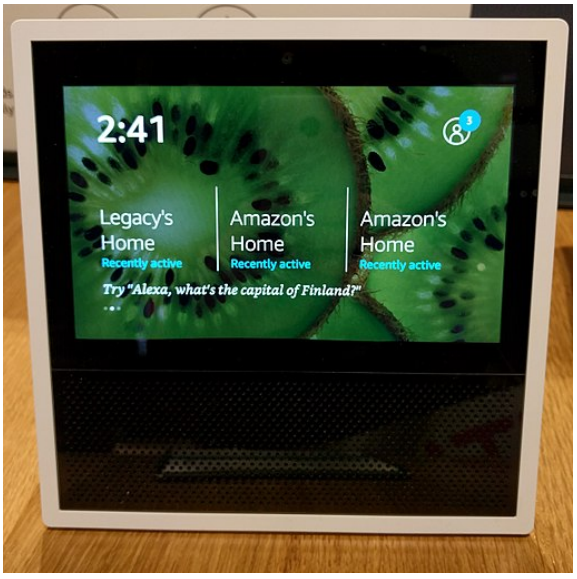

Figura 4: Dispositivo Echo Show con Alexa.
Fuente: https://commons.wikimedia.org/wiki/File:Amazon_Echo_Show_in_white.jpg

Con un ligero toque en la parte derecha del casco podremos visualizar a través de estas cámaras nuestro entorno exterior de forma que podremos recolocarnos.

Además, **Oculus Quest 2** es un dispositivo *standalone* (autónomo) por lo que no requerimos de un ordenador externo para funcionar, aunque opcionalmente lo podemos conectar mediante un cable (*Oculus Link*) o incluso de manera inalámbrica a través de *Virtual Desktop* [5].

También cabe destacar otros cascos VR que cuentan con capacidad similar como puede ser **HTC Vive**, que posee la tecnología *Lighthouse*[6] la cuál permite el rastreo del casco mediante 2 dispositivos que llenan la habitación de luz láser, para así detectar cualquier elemento de rastreo que cambie su posición en esa zona. **HTC Vive** tiene un precio mucho más elevado que **Oculus Quest 2**: 1010€. Otro ejemplo es **Valve Index** que posee también tecnología de seguimiento láser y destaca por poseer pantallas muy potentes con una tasa de refresco de 144 Hz, un campo de visión de 130º y tecnología IPS (el precio es de 1080€).

2.3. Leap Motion y Ultraleap

El laboratorio SIC adquirió en 2018 el dispositivo **Leap Motion**, el cuál nos sirve en este proyecto para el reconocimiento de manos y así lograr la ausencia de mandos físicos. Durante 2019, la empresa *Leap Motion* fue comprada por *Ultrahaptics* y la fusión se denominó como *Ultraleap*.

A partir de esta etapa, se fueron introduciendo otra serie de dispositivos, como el **STRATOS Inspire** (véase fig. 6). Este tiene la interesante función de simular el contacto físico con elementos virtuales a través de ultrasonidos. Además, añade una gran cantidad de sensores para detectar mejor los movimientos de nuestra mano en cada momento. Aunque este dispositivo es interesante y más potente que el **Leap Motion**, no es una buena opción para este proyecto, ya que la idea es encontrar un sensor portátil, pequeño y que además consiga un precio reducido (hay que remarcar que **Leap Motion** cuesta 100€ frente a los 3750€ que cuesta **STRATOS Inspire**).

Figura 5: Oculus Rift CV1 y Oculus Quest 2 respectivamente.
Fuentes:
https://commons.wikimedia.org/wiki/File:Oculus-Rift-CV1-Headset-Front.jpg y
https://commons.wikimedia.org/wiki/File:Oculus_Quest_2_-_2.jpg

2.4. Trabajos similares destacados

La Realidad Virtual es una tecnología joven y cara (un casco de realidad virtual junto con el equipo necesario para soportarlo puede rondar los 1200€), esto ha producido que no sea accesible para todos los públicos y por tanto, que no se hayan realizado demasiados proyectos de gran importancia.

Incluso en el ámbito de los videojuegos, tras años desde la primera aparición de los cascos VR, siguen siendo escasos los títulos que poseen una gran calidad o utilizan de manera eficiente la tecnología. Esto se debe a lo mencionado anteriormente, si el número de usuarios con un casco VR es pequeño, los desarrolladores corren más riesgo de que su aplicación sea un fracaso. Es decir, hasta que esta tecnología no sea más **integrada en la sociedad**, no comenzarán a desarrollarse mejores aplicaciones.

En esta sección vamos a discutir proyectos que han sido realizados en los últimos años que tienen una cierta similitud o utilizan las tecnologías de una forma parecida a lo que estamos realizando en este trabajo.

Figura 6: STRATOS Inspire, dispositivo de Ultraleap.
Fuente: https://www.ultraleap.com/product/stratos-inspire/

2.4.1. Future Sport VR

Future Sport VR[7] es un proyecto creado por *Kevin Viet Le* y subido a la plataforma *Hackster.io*. Este usa las mismas tecnologías que nuestro trabajo: Oculus Rift, Leap Motion y Alexa.

El proyecto consigue crear un entorno virtual similar a un salón o habitación multimedia (véase fig.7) donde nos encontramos diversas pantallas, en ellas podremos arrastrar vídeos o retransmisiones en directo. Además, a través de Alexa, se puede obtener datos sobre deportistas o incluso comprar entradas de eventos deportivos.

Al igual que este proyecto, *Future Sports VR* fue desarrollado en Unity, para ello utilizaron el *standard asset* del mismo. Para la parte de Leap Motion usaron los controladores de Orion software y en lo que respecta a Alexa, utilizaron AWS (para poder comunicar Unity y Alexa), la API de Twitter (para introducir los vídeos en alguna de las pantallas) y la API de Sportsradar (nos permite conocer datos de los jugadores y comprar entradas a eventos deportivos).

2.4.2. Control your "Earth Rover" in Virtual Reality

Control your "Earth Rover" in Virtual Reality[8] es un proyecto alojado en *Hackster.io* y creado por *Ron Dagdag*. Las tecnologías que utiliza son: Oculus Rift, Leap Motion, una cámara y un coche radiocontrol.

Figura 7: Pantalla principal de *Future Sports VR*.
Fuente:[7]

El proyecto consiste en "simular" al Mars Rover (vehículo que fue lanzado a Marte para tareas de exploración) con un coche radiocontrol y una cámara conectada. A partir de aquí, mediante Oculus Rift podemos visualizar las imágenes que capta el vehículo radiocontrol y con nuestras manos (Leap Motion) controlaremos el movimiento del mismo (véase fig.8).

Especificando más sobre el desarrollo del proyecto, tenemos que indicar que el coche radiocontrol utilizado se llama "Rover Bot" y trae controladores para los motores, servomotores, ruedas, base, batería, sensores de distancia y una placa Arduino UNO. Aunque en el desarrollo el autor retiró la placa Arduino UNO y la reemplazó por una Intel Edison, además de colocarle la cámara necesaria para el proyecto (véase fig.9).

Pasando ahora al desarrollo de la aplicación, el autor decidió prescindir de Unity para así poder encontrar un *framework* que fuera sencillo y efectivo, por lo que acudió a **WebVR** (actualmente reemplazado por **WebXR**[9]). Este permite a los usuarios crear infinitas aplicaciones que son usadas y cargadas en la web simplemente conectando nuestro dispositivo de realidad virtual al ordenador y accediendo a la dirección asignada. Por tanto, encontró un módulo dentro de **WebVR** que soporta Leap Motion y utilizando una estructura en la nube para comunicarse con Intel Edison pudo controlar el coche radiocontrol, lo que le permitió finalizar el desarrollo de la aplicación.

2.4.3. Games SDK for Alexa

Games SDK for Alexa[10] es un asset (recurso) lanzado a la tienda de Unity en 2018 por Austin Wilson. Este asset proveía un framework para desarrollar un proyecto en Unity y una skill de Alexa, de forma que nosotros mismo podemos definir el comportamiento que la skill tendrá sobre Unity.

El asset incluía archivos para Unity con funciones a cumplimentar y un tutorial en su web para configurar de forma adecuada nuestra skill de Alexa. Este proyecto utiliza AWS y PubNub (una empresa dedicada a los servicios en la nube), a diferencia de nuestro trabajo que utiliza AWS y CloudAMQP (colas almacenadas en la nube).

En el vídeo de ejemplo de su página web nos muestra un videojuego de plataformas llamado *The Explorer* y el usuario le pregunta a Alexa qué elemento se encuentra enfrente de él; esta le indica lo que se encuentra enfrente de su personaje (rocas, enemigos, étc.). Tras esto, el usuario le pide a Alexa que realice un ataque y el personaje de *The Explorer* rompe la roca y ataca a los enemigos.

Aunque a primera vista parece bastante interesante, apenas tuvo éxito y las pocas personas que adquirieron el asset en la tienda de Unity (el precio es 4.47€) han presentado quejas y fallos de varios tipos. Es por ello que el propio desarrollador ha calificado su asset en la tienda como *Deprecated* (obsoleto) y se ha ofrecido a devolver el dinero a los que lo adquirieron. Además, indica que está trabajando en una nueva versión.

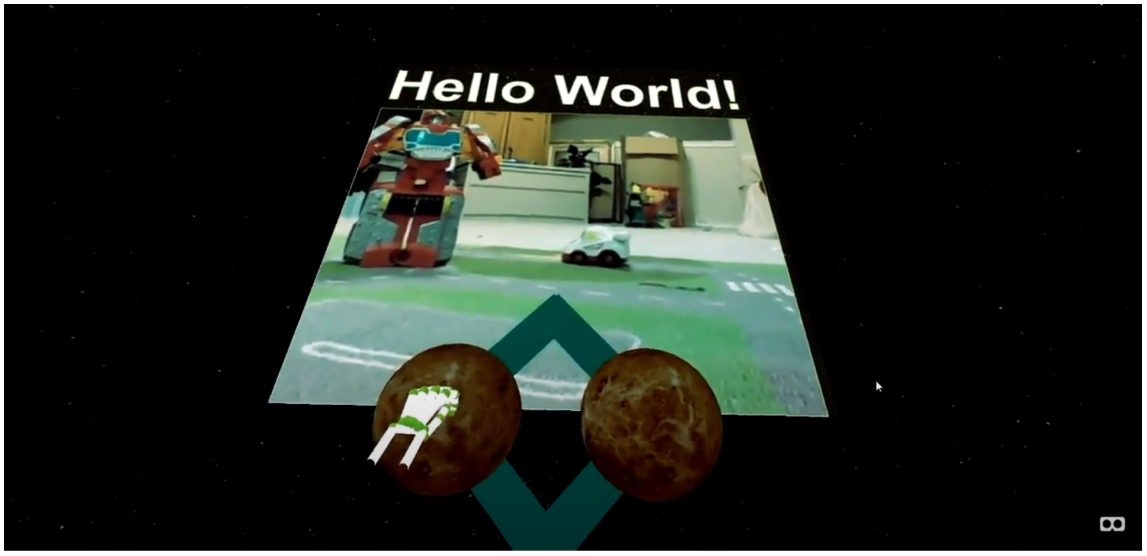

Figura 8: *Control your "Earth Rover" in Virtual Reality.*
Fuente:[8]

Figura 9: Rover Bot final.
Fuente:[8]

3. Fundamentos Teóricos

En esta sección profundizaremos en los conceptos que componen a cada una de las tecnologías utilizadas para así familiarizarnos con los límites y capacidades que poseen cada una de ellas.

3.1. Realidad Virtual

La Realidad Virtual se define como la creación de escenas con objetos de apariencia real y que sumergen al usuario en las mismas. Para ello, por supuesto, es necesario la utilización de sistemas informáticos como los **cascos de Realidad Virtual** que nos permiten visualizar susodichas escenas.

El término **Realidad Virtual** nació en la década de los 80 popularizado por **Jaron Lanier**, que fue de las primeras personas en investigar en esta tecnología. Lanier además fundó **VPL Research, Inc.** en 1985, que fue la primera empresa que comercializó cascos y guantes de realidad virtual[11] (véase fig.10).

Los cascos de Realidad Virtual actuales utilizan sobre todo tecnología de giroscopio (para detectar la orientación en la que miramos) y sensores externos para la posición de nuestra cabeza, aunque recientemente se está comenzando a incluir cámaras de vídeo en el propio casco que podrían sustituir a estos sensores. En cuanto a la pantalla, el mínimo de fotogramas por segundo debe ser de 60 para evitar mareo y la calidad de imagen suele rondar el FullHD(1920x1080).

En la Realidad Virtual son muy importantes dos conceptos: **presencia** e **inmersión**. El primero de ellos se refiere al hecho de estar presente junto a otra serie de objetos o seres y el segundo al grado de realismo (o verosimilitud) que posee el entorno, de forma que el usuario piense que los objetos son iguales a los del mundo real.

3.1.1. Tipos de Realidad Vrtual

Existen varios tipos de realidad virtual en función del grado de **inmersión** que posea la aplicación: **inmersiva**, **no inmersiva** o **semi-inmersiva**.

La Realidad Virtual **inmersiva** consiste en un escenario 3D que visualizamos mediante un casco VR (o similar) y con el que podremos interactuar a través del movimiento de nuestras extremidades, la dirección en la que miramos, étc. Nuestro proyecto entra dentro de la definición de Realidad Virtual **inmersiva**.

La Realidad Virtual **no inmersiva** se visualiza a través de un ordenador donde podemos interactuar con una gran variedad de entornos. Es decir, la realidad virtual **no inmersiva** no se relaciona con los cascos VR sino con la interacción que nos puede ofrecer Internet. Ejemplo de ello es la navegación en webs o los videojuegos.

Figura 10: Jaron Lanier en VPL Research, 1988.
Fuente: https://professorstevenskovholt.com/2014/06/10/the-professor-queries-jaron-lanier-in-1988/

Debido al alto precio de ciertos dispositivos VR inmersivos, la realidad virtual **no inmersiva** ha sido siempre la base a la hora de crear aplicaciones virtuales.

La Realidad Virtual **semi-inmersiva** es similar a la **inmersiva** con la diferencia que la forma de visualización no es un casco VR, sino 4 pantallas (o proyectores en cada pared) que nos muestran un entorno virtual. La ventaja de esto es que podemos interactuar con objetos del mundo real (podemos incluir volantes, herramientas...). Aunque no dispone de un casco VR, es necesario unas gafas y algunos dispositivos de seguimiento, ya que contiene un cierto grado de inmersión y por tanto deberemos mantener algún medio para interaccionar con los componentes virtuales. El ejemplo más popular es *Cave Automatic Virtual Environment* (véase fig.11) que fue creado en 1991 en la Universidad de Illinois (Chicago) utilizando proyectores, una sala oscura y una serie de espejos colocados adecuadamente.

3.1.2. Métodos de Realidad Virtual

La realidad virtual puede orientarse hacia diversos métodos según el ámbito para el que se quiera desarrollar. Aunque la base siga siendo la misma, el enfoque que tomará

Figura 11: Cave Automatic Virtual Environment.
Fuente: https://commons.wikimedia.org/wiki/File:CAVE_Crayoland.jpg

el entorno o aplicación debe cambiar, para así conseguir un mayor rendimiento del mismo. Los métodos de Realidad Virtual principales son:

- **Simuladores.** El objetivo de estos es crear un entorno realista que los usuarios puedan usar como práctica o estudio en diversos ámbitos. Para ello se requiere una buena inmersión y una interacción eficiente entre el entorno y el usuario (ya que la intención es imitar de la manera más fiel posible el mundo real). El proyecto que estamos tratando forma parte del método **simuladores**.

- **Avatares.** Con los avatares los usuarios pueden crear un personaje eligiendo la apariencia que deseen (la intención es escoger una similar a la propia) y entonces interactuar en el entorno con el resto de usuarios, de esta manera podemos crear reuniones, hacer conferencias, étc. El ejemplo más popular de esto es *VRChat* (véase fig.12).

- **Proyección de imágenes reales.** Consiste en la proyección de imágenes o entornos reales (ambos normalmente generados por ordenador) y suele estar aplicado en simuladores de vuelo o navegación autónoma.

- **Por ordenador.** Como comentamos anteriormente en la **realidad virtual no inmersiva**, este método consiste en mostrar un mundo 3D a través de la pantalla de nuestro ordenador, sin ningún tipo de sensor que nos permita un mayor grado de inmersión como lo hacemos en la **realidad virtual inmersiva**.

Figura 12: VRChat.

Fuente: https://aiaustin.wordpress.com/2014/12/03/vrchat-experiment/

- **Inmersión en entornos virtuales.** Corresponde a la **realidad virtual inmersiva** y por tanto debe utilizar sensores que nos permitan comunicar nuestro estado corporal (o cerebral) directamente con un dispositivo externo. Ejemplo de esto serían los cascos VR o los sensores de latidos del corazón.

3.2. Unity

Unity (véase fig.13) es un motor de videojuego multiplataforma lanzado en Mayo de 2005 que funcionaba exclusivamente en equipos Mac (Apple), sin embargo, con el éxito obtenido pudo expandir sus herramientas y capacidades para así llamar la atención de estudios más grandes e incorporarse al resto de equipos (Windows y Linux).

Su éxito se debe mayormente a la falta de recursos que poseen desarrolladores independientes para crear su propio motor de juego, por lo que Unity apareció en escena para subsanar esta situación y aumentar la accesibilidad a la hora de crear aplicaciones o videojuegos[12]. A raíz de esto y las mejoras que se fueron introduciendo, desarrolladores de todo tipo pusieron su atención en este motor.

Figura 13: Pantalla principal de Unity.
Fuente: https://unity3d.com/beta/2020.2b

3.2.1. Características principales

Unity posee soporte para **Blender**, **Adobe Phostoshop**, **3ds Max** y un gran número más de aplicaciones. Con los archivos de estas, creamos los llamados *prefab* que forman un tipo de objeto a partir del cuál se crearán las instancias necesarias del mismo. Esto es bastante útil ya que los cambios que realicemos en el *prefab* se aplicarán a cada una de las instancias que tengamos creadas, pero los cambios a cada una de las instancias sólo se mantendrán en la misma.

Cabe mencionar que Unity cuenta con soporte para Realidad Virtual (Oculus, HTC Vive, Valve...), Realidad Aumentada, mapeado de relieve y reflejos, sombras dinámicas, efectos de post-procesamiento e incluso soporte para *Nvidia* y el motor de física *PhysX* entre muchas otras características. Además, Unity nos permite crear aplicaciones para Linux, Windows, MacOS, táblets y smartphones e incluso para consolas como *Nintendo 3DS*[13]. Es por estas razones que podemos considerar a Unity una plataforma muy flexible.

En cuanto al código, Unity se basa en *Mono*, la implementación de código libre de *.NET Framework* (el framework de Microsoft). Por ello, el lenguaje predeterminado que posee Unity es **C#**, pero puede utilizarse **Boo** (similar a *Python*) o un lenguaje personalizado llamado **UnityScript**.

Unity además utiliza la tecnología *Mecanim* (desarrollada por el propio Unity) para

el control de las animaciones. *Mecanim* permite un control fluido e intuitivo de las mismas y sus transiciones, e incorpora árboles de estado para una creación más sencilla.

Por último, mencionar que Unity cuenta con una tienda propia integrada en su interfaz llamada *Asset Store*. En ella, los usuarios pueden subir o descargar *asstes* (paquetes de Unity con distintas herramientas, elementos y objetos) a un cierto precio o gratuitos.

3.2.2. Licencias

Unity tiene un modelo propio a la hora de obtener beneficios y este se basa en el uso de distintas licencias:

- **Unity Personal.** Es gratuita y no tiene límites en cuanto a la capacidad del programa, pero las aplicaciones que creemos tendrán una marca de agua con "Made with Unity" al inicio de la misma. Por otro lado, el umbral de ingresos con las creaciones de esta licencia es de *100.000 dólares*. Si superamos este umbral, deberemos adquirir la licencia **Unity Plus** o **Unity Pro**.
- **Unity Plus.** Tiene un precio de *400 dólares/año* y está enfocado sobre todo al desarrollo de juegos o aplicaciones móviles. Este servicio tiene ciertas limitaciones en las capacidades de Unity y posee un umbral de *200.000 dólares* en las ganancias de las aplicaciones que creemos. Si superamos este umbral, deberemos adquirir la licencia **Unity Pro**.
- **Unity Pro.** Esta es la licencia definitiva de Unity. No posee ningún tipo de limitación a la hora del desarrollo o uso de la interfaz, no tiene un umbral de ingresos y permite hasta 200 usuarios conectados a la vez en *Unity Multiplayer* (servidores proporcionados por Unity que permiten el juego entre varios jugadores). El precio es de *1800 dólares/año*.
- **Otras licencias.** Además de las principales, Unity posee una serie de licencias especiales como **Unity Enterprise**, orientado a que las empresas lleguen a un acuerdo con Unity para que sus trabajadores puedan tener una experiencia personalizada del mismo por un precio acordado mensualmente, o **Unity Education** orientado a estudiantes.

3.2.3. Unity, Oculus SDK y Unreal Engine

Unity es el *motor* elegido con el que se modelará nuestra aplicación virtual, con este definiremos interacciones, elegiremos el comportamiento del sistema y programaremos los *scripts* necesarios. Sin embargo, a la hora de programar un sistema virtual con un casco VR como es **Oculus Rift CV1**, barajamos previamente otras 2 opciones con gran potencial.

La primera de ellas es **Oculus SDK**, el *kit de desarrollo* proporcionado por Oculus para crear aplicaciones virtuales con una gran eficiencia en sus cascos VR. Este es un SDK abierto compatible con Windows, Linux y MacOS y soporta C++. Esta forma de crear una aplicación virtual se le conoce como *nativa* ya que programamos directamente en C++ utilizando las distintas librerías añadidas por *Oculus SDK* y sin un editor visual que nos muestre la aplicación en cada momento, por lo que deberemos configurar nuestro propio entorno de desarrollo. Sin embargo, poseemos archivos (*assets*) que nos permiten trasladar parte de las características de **Oculus SDK** a motores de videojuegos como son **Unity** o **Unreal Engine**, por si preferimos otro entorno de desarrollo distinto.

La segunda opción fue **Unreal Engine**, un motor de videojuegos de gran potencial y destacado por su calidad de gráficos. Este motor suele ser utilizado para videojuegos AAA (videojuegos de primera clase que conllevan un gran desarrollo) y simulaciones realistas en tareas como la arquitectura o incluso el cine. Está programado en C++ y es el lenguaje soportado para el desarrollo, además posee un entorno de desarrollo visual (como Unity) que nos permite observar la aplicación en cada momento. Debido a estas grandes capacidades que posee, Unreal también es conocido por su complejidad y por ser un motor de videojuegos orientado a usuarios expertos o desarrolladores ya iniciados en el ámbito de las aplicaciones virtuales.

Por tanto, barajando estas 3 opciones: *Nativo, Unreal y Unity* decidimos finalmente **Unity** por varias razones. La primera de ellas es que al ser este proyecto iniciado como *alumno colaborador*, empezamos desde cero en este ámbito de conocimiento, por lo que utilizar un **entorno visual** nos permitiría de manera más rápida y sencilla comprender muchos conceptos. Es por ello que decidimos descartar *la manera nativa* (*Oculus SDK*) para así facilitar el proceso de aprendizaje. Otra de las razones es que, como hemos comentado anteriormente, *Unreal Engine* es un motor de juego complejo orientado a aplicaciones virtuales de gran detalle y potencial gráfico. En nuestro caso, en un primer momento el objetivo era crear un entorno sencillo que pudiera aunar las tecnologías propuestas, por lo que no necesitamos de una gran potencia gráfica y preferimos buscar la opción que nos permita aprender los conceptos básicos de manera más rápida, para así una vez conseguido los objetivos iniciales profundizar en el entorno de desarrollo seleccionado. Por tanto, decidimos escoger Unity, debido a la facilidad y rapidez que lo caracterizan, para así poder crear una aplicación virtual con un potencial gráfico decente mientras mantenemos una curva de aprendizaje muy alta.

3.3. Oculus VR

Oculus VR se fundó en Julio de 2012 y es una empresa centrada en el desarrollo de software y hardware de Realidad Virtual. Esta comenzó en abril de 2012 cuando anunciaron su primer prototipo de casco VR a través de una campaña de *Kickstarter*

la cuál tuvo éxito y permitió que se desarrollaran 2 modelos: *Oculus Rift Development Kit 1* (el cuál podemos ver en la fig.14) y *Oculus Rift Development Kit 2*.

Figura 14: *Oculus Rift Developmente Kit.*
Fuente: https://commons.wikimedia.org/wiki/File:Oculus_Rift_-_Developer_Version_-_Front.jpg

En 2014 **Oculus VR** fue adquirida por **Facebook**[14], en 2015 **Oculus VR** se alió con **Samsung** para crear *Gear VR*[15], un casco VR que funciona a través de nuestro *smartphone* Samsung como podemos ver en la fig.15 y finalmente, en 2016 se lanzó al mercado la primera generación de productos **Oculus VR**: *Oculus GO*, *Oculus Quest* y *Oculus Rift CV1* (véase fig.16).

3.3.1. Primera Generación de Oculus

La primera generación se puede definir de 2016 a 2019 y compone los 3 primeros productos que **Oculus VR** lanzó siendo propiedad de **Facebook**:

- **Oculus Rift**. El primero en lanzarse a principios de 2016, se vendió conjuntamente (y de forma separada también) con los *Oculus Touch* (los controladores de Oculus que poseen botones y paneles táctil, además de rastreo de posición). Este posee pantallas con una resolución de 1080x1200 para cada ojo y corre a 90hz, además posee altavoces integrados. Nos permite movernos unos metros (limitado por nuestra habitación u oficina) gracias a los sensores que trae los cuáles detectan la posición de nuestra cabeza y manos (*Oculus Touch*) en cada momento. Esto se lleva a cabo mediante una serie de emisores infrarrojos

Figura 15: *Gear VR*.
Fuente: https://commons.wikimedia.org/wiki/File:Samsung_Gear_VR.jpg

colocados en el casco VR que emiten un patrón distintivo que los sensores son capaces de captar y por tanto definir la posición actual de nuestra cabeza.

- **Oculus GO**. Fue el segundo dispositivo en lanzarse y lo hizo a principios de 2018. Es un dispositivo *standalone* (autónomo) ya que no requiere de un ordenador externo para funcionar. Posee un controlador táctil y con algunos botones y se basa en un sistema operativo móvil (Android). Fue inicialmente vendido por 200/250 dólares según si elegimos la versión de 32 o 64 GB de capacidad. Posee una pantalla de 1280x1440 en cada ojo y tiene una tasa de refresco de 60 a 72hz, esto junto al hecho de que es un dispositivo no demasiado potente hace que en ocasiones las simulaciones o juegos que poseen mayor movimiento produzcan mareo o fatiga. Las aplicaciones podían ser descargadas desde la tienda nativa de Oculus (mediante el propio casco) o instalarlos desde nuestro PC por cable (micro-usb).

- **Oculus Quest**. El último dispositivo en sumarse a esta generación fue *Oculus Quest* a mediados de 2019. Pretendía ser un sucesor de *Oculus GO* que podría ser utilizado de manera autónoma pero con mayor capacidad y potencia. Era incluso compatible con *Oculus Touch*, los controladores de *Oculus Rift*. Posee una serie de cámaras que además de rastrear a estos controladores, nos sirven

Figura 16: *Oculus GO*, *Oculus Quest* y *Oculus Rift CV1* respectivamente
Fuentes: https://commons.wikimedia.org/wiki/File:Oculus_Go_-_7.jpg, https://commons.wikimedia.org/wiki/File:Oculus_Quest.jpeg y https://commons.wikimedia.org/wiki/File:Oculus-Rift-CV1-Headset-Front.jpg

para ver el entorno que nos rodea y colocarnos de manera correcta para evitar así golpes o situaciones donde nos hemos excedido en el movimiento dentro de la aplicación. Tiene una pantalla OLED de 1440x1600 en cada ojo y corre a 72hz. Además, se basa al igual que en el caso de *Oculus GO* en un sistema operativo Android. En una actualización posterior a su lanzamiento, *Facebook* lanzó el **Oculus Link**, un cable USB que nos permite utilizar *Oculus Quest* conectado a nuestro ordenador y disfrutar del software compatible con *Oculus Rift*. Por último, en Mayo de 2020 se añadió soporte para usar software de reconocimiento capaz de detectar nuestras propias manos (usando sus cámaras integradas), es decir, utilizar las aplicaciones sin mandos.

3.3.2. Segunda Generación de Oculus

Poco antes de la salida de *Oculus Quest*, se anunció *Oculus Rift S*, el sucesor de la versión original del mismo, y al poco de su anuncio, el casco VR original dejó de venderse, aclarando la compañía que pretende centrarse en este nuevo casco y por tanto, desde el lanzamiento de *Oculus Rift S*, podemos marcar el comienzo de esta **segunda generación**.

Esta generación, la más reciente, se forma por **Oculus Rift S** y **Oculus Quest 2** (véase fig.17), sin embargo, tenemos que mencionar que según unas declaraciones recientes de la compañía, su intención es centrarse en **Oculus Quest 2** y dejar de

comercializar la versión Rift S, ya que el primer casco tiene todas las capacidades del segundo (e incluso más) y no necesita de conexión al ordenador para funcionar (es *standalone*).

Oculus Rift S **Oculus Quest 2**

Figura 17: *Oculus Rift S* y *Oculus Quest 2* respectivamente
Fuentes: *https://commons.wikimedia.org/wiki/File:Oculus_rift_s.jpg* *y* *https://commons.wikimedia.org/wiki/File:Oculus_Quest_2_-_2.jpg*

Describiendo en profundidad esta generación:

- **Oculus Rift S**. Lanzado en Marzo de 2019, comenzó con un precio de 400 dólares. Posee una pantalla de 1280x1440 para cada ojo a 80hz. Tiene 5 cámaras (2 a los lados, 1 arriba y 2 abajo) para rastreo de los mandos y nuestra posición. Tiene los mismos requisitos de hardware y software que la versión CV1. Posee altavoces integrados y la capacidad de ver a través de las cámaras nuestra posición en la vida real para recolocarnos correctamente. Como hemos comentado anteriormente, el 16 de Septiembre de 2020, Facebook anunció que dejaría de producir este modelo a partir de la primavera de 2021 para así dar prioridad a la producción y desarrollo de **Oculus Quest 2**.
- **Oculus Quest 2**. Fue lanzado el 13 de Octubre de 2020, su precio de salida fue de 300 o 400 dólares según si elegimos la versión de 64GB o de 256GB. Su capacidad es de 6GB de RAM y tiene un procesador Qualcomm Snapdragon XR2. Posee 4 cámaras y una pantalla de 1832x1920 para cada ojo a 90hz. Este casco tiene la capacidad de ser autónomo (*standalone*) aunque con opción a conectarse a nuestro ordenador mediante el *Oculus Link* (un cable de tipo C de 5 metros), por lo que podríamos ejecutar las mismas aplicaciones que las versiones CV1 o Rift S. Además, con una actualización que se incluyó en

las **Oculus Quest 1**, soporta el rastreo de manos a través de las cámaras que incorpora, por lo que es un dispositivo bastante potente y versátil que demuestra por qué *Facebook* quiere apostar por él en los próximos años.

Figura 18: *Leap Motion*
Fuente: https://commons.wikimedia.org/wiki/File:Leap_Motion_Orion_Controller_Plugged.jpg

3.4. Ultraleap

Leap Motion fue una empresa que comercializaba un dispositivo de rastreo de manos y dedos que además utiliza estos como *inputs* (sin necesidad de contactos físicos) para las aplicaciones desarrolladas con el mismo. Su producto fue llamado de la misma manera, como podemos ver en la fig.18.

Esta tecnología se desarrolló en 2008 cuando *David Holz* (cofundador) estudiaba su doctorado en Matemáticas; años más tarde en 2010, fundó la compañía junto con *Michael Buckwald*. Tras varios años de obtener financiación, en 2013 comenzó a comercializarse el dispositivo a gran escala bajo el nombre de *The Leap*[16]. Inicialmente el software de desarrollo lanzado para este producto era pobre y no tenía demasiado soporte. En agosto de 2014 se introdujo un modo de rastreo en VR, pensado para colocar el dispositivo en un casco VR, aunque tuvieron que pasar casi 2 años para que la compañía decidiera sacar el software **Orion**, pensado exclusivamente para el desarrollo en Realidad Virtual. A partir del momento en el que la empresa decidió introducirse en el mundo de la Realidad Virtual, *Leap Motion* lanzó un adaptador (con un precio de 30€) para colocar su dispositivo en un casco VR como podemos ver en la fig.19. Además desarrolló varias demos interactivas y

Figura 19: Adaptador de *Leap Motion* para cascos VR
Fuente: https://www.ultraleap.com/product/vr-developer-mount/

juegos cortos donde podíamos experimentar completamente el potencial de ambas tecnologías juntas.

A pesar de la potencia que posee este dispositivo y de tener un precio relativamente bajo (100€), no ha tenido demasiado éxito en ventas y el desarrollo de sus aplicaciones sigue siendo algo experimental o inmaduro.

Esta es una de las razones por las que en 2019, *Ultrahaptics* (la empresa rival de Leap Motion) compró la compañía y las fusionó en **Ultraleap**.

3.4.1. Tecnología

Leap Motion es un periférico USB diseñado inicialmente para ser colocado en una superficie mirando hacia arriba y más tarde adaptado para el uso en vertical colocado en un casco VR. Usa 2 cámaras **monocromáticas infrarrojas** y 3 **LED's infrarrojos**. Este cubre una superficie hemisférica de hasta 1 metro. El funcionamiento consiste en el siguiente: los LED's envían luz infrarroja aleatoria y las cámaras recogen toda la información que se ha reflejado. Esta información se envía por USB al dispositivo y el software de *Leap Motion* analiza y define todo lo recibido[17].

El contenido o funcionamiento del software es un misterio no revelado por la com-

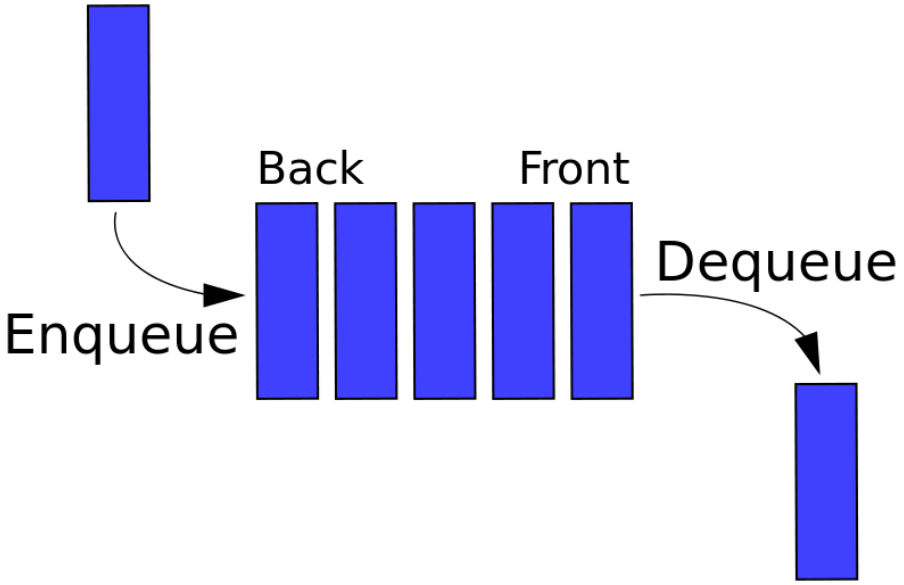

Figura 20: Representación de la estructura de datos **cola**
Fuente: ***Vegpuff/Wikipedia****:*
https://commons.wikimedia.org/wiki/File:Data_Queue.svg

pañía pero consigue con mucha precisión y calidad rastrear nuestras manos y dedos. Tanto es así que el propio software para Realidad Virtual, **Orion**, nos proporciona una serie de botones, deslizadores y objetos con los que podemos interactuar sin problemas de una manera fluida y natural.

Un inconveniente arraigado a la propia tecnología del dispositivo, que hemos podido comprobar en la realización de este proyecto, es que al no tener las manos delante del dispositivo (en VR es delante del casco) estas dejarán de existir en el entorno virtual. Es decir, si queremos lanzar una pelota en una aplicación de Unity utilizando un casco VR, debemos mantener nuestra cabeza mirando a las manos en todo momento del lanzamiento. La solución a esto sería colocar un dispositivo estático que fuera capaz de rastrear nuestras manos y que además la rotación de nuestro cuerpo no fuera un inconveniente.

3.5. Cola de Mensajes

En informática, una **cola** se conoce como una *estructura de datos* donde sus elementos siguen el orden de una fila, ya que el primero en entrar (mediante operación *push*) también será el primero en salir (mediante una operación *pull*) como podemos ver en la figura 20. En inglés esto se conoce como **FIFO** (*First In, First Out*).
Partiendo de esto, podemos describir una **cola de mensajes** como un sistema donde distintas aplicaciones (o incluso los propios procesos de una misma aplicación) envían

y recogen información de una **cola**. Es decir, una **cola de mensajes** permite a las aplicaciones almacenamiento temporal de información hasta que la aplicación receptora deje de estar ocupada.

En la práctica puede ser similar a una **API REST**, sistema utilizado para comunicar aplicaciones a través de HTTP. Sin embargo, el uso de HTTP no es necesario en la **cola de mensajes** (aunque es posible implementarlo) y las aplicaciones no consumen tantos recursos en la recepción del mensaje ya que la **cola intermediaria** permite que estas recojan el mensaje cuando estén disponibles de nuevo.

A raíz de aquí podemos introducir **AMQP**: **Advanced Message Queuing Protocol** (Protocolo de Cola de Mensajes Avanzado). Este es un protocolo estándar abierto que permite la transmisión de mensajes en la capa de Aplicación junto con otra serie de operaciones como *mensajes orientados con garantía de entrega* o *autenticación y/o encriptación con SAS y/o TLS*[18].

3.5.1. RabbitMQ y CloudAMQP

RabbitMQ es un software que funciona como *middleware* en mensajería. Es decir, actúa como un intermediario a la hora de comunicar mensajes entre aplicaciones. Este implementa el estándar **AMQP** y posee API para *Java*, *.NET* (C#) y *Python*.

Por otro lado, si se quiere buscar una opción de **cola de mensajes en la nube** que use **RabbitMQ**, disponemos de **CloudAMQP**, el cuál nos permite contratar distintos planes mensuales según nuestro uso del servicio y nos proporciona documentación en cada uno de los lenguajes soportados por **RabbitMQ**, para así poder realizar aplicaciones que contengan un *middleware* en la nube.

3.6. Alexa y Amazon Web Services (AWS)

Alexa (o **Amazon Alexa**) es un **asistente virtual con inteligencia artificial** desarrollado por Amazon. A través de la interacción por voz es capaz de reproducir música, poner alarmas, crear listas de tareas, proveer información sobre el tiempo o el tráfico, étc. Incluso podemos controlar dispositivos de nuestra *smarthome* como bombillas, enchufes o la televisión.

Para crear una experiencia personalizada con Alexa, Amazon pone a disposición de los usuarios las *skills* (aplicaciones desarrolladas para Alexa) que nos permiten realizar una serie de acciones definidas por el desarrollador. Con estas podemos hacer un sinfín de acciones más allá de las que trae por defecto Alexa; por ejemplo, podemos instalar una *skill* que nos permita consultar recetas de cocinas o aprender palabras en otros idiomas. Para ello, los desarrolladores disponen del **Alexa Skill Kit** (ASK), una API en la nube con documentación y ciertas pautas para que podamos diseñar

Figura 21: Familia de productos *Alexa* o *Echo*
Fuente: https://mundocontact.com/arma-tu-sistema-de-sonido-en-casa-con-echo-y-alexa-sin-cables/

un modelo de conversación y además introducir nuestro propio código (en *Javascript* o *Python*)[19].

A partir de 2019, Amazon lanzó una serie de productos nuevos añadiéndoles tecnologías adicionales como puede ser el *Echo Show* (con pantalla) o el *Echo Studio* (con altavoz *Dolby sound*), véase fig.21

Amazon permite a los distribuidores de dispositivos integrar las capacidades de Alexa usando el *Alexa Voice Service* (*AVS*), una API basada en la nube que proporciona a los desarrolladores **reconocimiento del habla** (ASR) y **comprensión natural del lenguaje** (NLU)[20]. Además, en 2016, Amazon anunció que haría que la tecnología detrás de ASR y NLU estuviera disponible para desarrolladores bajo el nombre de **Amazon Lex**. Este permitiría a los creadores hacer un *chatbot* entre los usuarios y el sistema, similar a lo que ya es *Alexa*[21].

A partir de aquí vamos a discutir los aspectos más importantes de **Amazon Web Services**. AWS es una plataforma que provee diferentes **servicios de computación en la nube** para cualquier sector: individuales, compañías, gobiernos, étc. AWS consiste en *pagar por lo que utilices* (*pay-as-you-go*), de forma que activar distintos componentes tiene un precio por hora o incluso por minuto y al final del mes se calcula el tiempo total de uso.

AWS fue lanzado en 2002 pero ofrecía servicios muy difusos y poco consistentes. El concepto se remodeló en 2003 cuando *Chris Pinkman* y *Benjamin Black* presentaron un documento describiendo una posible perspectiva del servicio tal y como se conocía. Declaraban que Amazon podría ofrecer un sistema de alquiler de servidores virtuales y que el cobro del tiempo usado permitiera sacar beneficio a la infraestructura montada[22]. Tras esto, en 2006, AWS fue re-lanzado oficialmente ofreciendo sus 3 servicios principales (*almacenamiento en la nube*, *SQS* y *EC2*) y trabajando en los conceptos que ambos *Chris y Benjamin* propusieron.

AWS ha declarado en 2020 poseer más de 175 servicios, aunque en este momento vamos a hablar de los 3 principales:

- **EC2**. *Amazon Elastic Compute Cloud* es un servicio que ofrece a los usuarios alquilar ordenadores virtuales en los cuáles ejecutar su código o aplicación. Para inicializar el equipo, Amazon pone en disposición de estos el *AMI* (*Amazon Machine Image*), de forma que puedan configurar sus **máquinas virtuales**: elegir capacidad del equipo, sistema operativo, servicios de red, parar, terminar, reanudar su ejecución, étc. A estas máquinas virtuales con las configuraciones elegidas por el usuario se les denominan **instancias**.

- **Lambda**. *AWS Lambda* es un servicio de computación basado en eventos y sin servidor, que permite a los usuarios correr nuestro código sin necesidad de preocuparnos por las capacidades del equipo. Es decir, **AWS Lambda** permite que un usuario suba su código y este se ejecute a la espera de una serie de eventos o estímulos; cuando esto suceda, el código ejecutará las acciones oportunas y el encargado de administrar los recursos que necesite nuestra aplicación será AWS. El objetivo de este servicio es simplificar la ejecución de un cierto código, ya que en muchas ocasiones montar un equipo virtual mediante EC2 será más costoso y requerirá más tiempo si nuestra aplicación a ejecutar es bastante sencilla.

- **SQS**. *AWS Simple Queue Service* es un servicio de **cola de mensajes** distribuido. Este soporta el envío de mensajes mediante *aplicaciones de servicio en la web* (HTTP, TCP/IP, Java, étc.). Está orientado a soportar una **gran escalabilidad** y solucionar problemas como el del *productor-consumidor*. Como hemos mencionado anteriormente, es uno de los primeros servicios introducidos desde el re-lanzamiento de AWS en 2006.

3.7. Inteligencia Artificial

La **Inteligencia Artificial** se define como la inteligencia expresada por máquinas. Este campo se conoce por el estudio de los *agentes inteligentes*, formado por cualquier dispositivo capaz de realizar una serie de acciones, dependiendo de su entorno, que consigan un acercamiento o la llegada a una meta u objetivo.

La IA ha sufrido distintas épocas de popularidad y decepción desde su creación como disciplina académica en 1955. Sin embargo, el principal culpable de que la IA esté actualmente en el punto de mira es *AlphaGO*, un programa de ordenador capaz de jugar al juego de mesa *Go!*, ya que en 2015 consiguió ganar a un jugador profesional del mismo.

Aunque la IA es cada vez más potente y utiliza técnicas mejores (como pueden ser el *machine learning*) sigue existiendo una serie de problemas inherentes a la misma:

- **Razonamiento**.
- **Representación del conocimiento**. Esto quiere decir que debemos modelar cada problema de una forma que sea *tratable* para el **agente inteligente**.
- **Aprendizaje**.
- **Procesamiento Natural del Lenguaje**. Definido como la forma que tiene una máquina de adaptar su lenguaje al de los seres humanos para que la comunicación sea más natural al usuario.
- **Percepción**.

3.7.1. Minimax

La IA tiene un sinfín de técnicas y algoritmos, pero en esta sección vamos a tratar el **Minimax**, ya que es el método utilizado en este proyecto. Añadir que la forma de implementación fue en un *juego de adversarios*, por lo que nuestra IA no solo debe tener en cuenta su movimiento, sino también los movimientos del adversario.

Antes de explicar el **algoritmo del Minimax**, debemos comentar que *MAX* corresponderá a la jugada del adversario y *MIN* corresponderá a la jugada de nuestra IA; además, los estados donde el **adversario gana** deberán recibir un **valor mayor** que los estados donde la **IA gana**, para así asegurar el correcto funcionamiento del algoritmo. Esta estrategia consiste en el uso del árbol de decisiones del juego, por lo que cada nodo poseerá un valor y este será conocido como el *valor Minimax*[23] (que escribiremos *MINIMAX(n)*) y representa cómo avanzará la partida a partir de ese nodo suponiendo que ambos jugadores realizan su jugada óptima.

El **algoritmo del Minimax** usa una computación simple y recursiva de los *valores Minimax* en los nodos sucesores. La forma de computarse este valor en cada uno de los nodos sigue la definición vista en la fig.22. Si nuestro nodo es terminal, entonces le asignaremos *Utility*, que será el valor que hemos decidido colocar a cada nodo final. Si no es terminal y el jugador actual es *MAX*, entonces elegiremos la jugada (nodo sucesor) cuyo *MINIMAX* sea **mayor** (aquí comienza la recursión). Por último, si el

nodo no es terminal y el jugador actual es *MIN*, entonces elegiremos la jugada (nodo sucesor) cuyo *MINIMAX* sea **menor**.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

Figura 22: Definición del valor Minimax
Fuente:[23]

El algoritmo funciona de la siguiente manera: la IA recibe el estado actual del juego (nodo actual) y debe obtener su **valor Minimax**. Si no es un estado final, tendrá que recibir el **valor Minimax de sus sucesores** para elegir el *mínimo* y así determinar el movimiento. Por tanto, comenzará a recorrer el árbol de decisiones suponiendo que cada jugador ha realizado su jugada óptima (*MAX* elige el mayor MINIMAX y *MIN* el menor). El *recorrido* continuará ejecutándose hasta que se llegue a estados terminales, cuando esto ocurra, el algoritmo *de manera recursiva* (como hemos comentado anteriormente) dará valores a los predecesores de los nodos terminales, hasta que llegue a los sucesores del nodo actual, con lo que ya podremos elegir un movimiento.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figura 23: Pseudocódigo de Minimax
Fuente:[23]

Podemos encontrar en la fig.23 el pseudocódigo correspondiente al método **Minimax**. La función *MINIMAX-DECISION* es la inicial ya que recibe un estado (nodo) y a continuación devuelve la jugada escogida por la IA. Para ello, utiliza la función *MIN-VALUE* que a partir del estado actual supone jugadas (hasta un estado terminal) donde ambos jugadores realizarán su movimiento óptimo; en el caso de esta función se elegirá la jugada con menor *MAX-VALUE*, lo que quiere decir que escogerá la jugada que menos beneficie al adversario. Para la jugada óptima del adversario (*MAX-VALUE*), este también tendrá en cuenta que sea la peor para la IA, esa es la razón por la que dentro de *MAX-VALUE* se escoge la jugada con mayor *MIN-VALUE*.

4. Desarollo

En esta sección se profundizará en las distintas etapas del desarrollo de manera cronológica, ya que la intención es mostrar la forma en que se fue construyendo este proyecto. Otra razón por la que se ha escogido esta forma de tratar el desarrollo es que antes de la realización del mismo no teníamos el conocimiento suficiente para utilizar muchas de las tecnologías descritas, por lo que parte del desarrollo fue también un aprendizaje de los conceptos básicos de cada una.

4.1. Material

El material utilizado para este proyecto fue el casco VR **Oculus Rift CV1** (véase fig.5), el dispositivo de rastreo de manos **Leap Motion** (véase fig.18), un **Amazon Echo Dot** (véase fig.32) y el ordenador del laboratorio SIC, el cuál posee las siguientes características:

- **RAM**: 32GB
- **CPU**: Intel Xeon 12 CPU $\sim$ 2.1GHz
- **Gráfica**: NVIDIA GeForce GTX 960 (VRAM: 4GB)
- **Almacenamiento**: 1TB SSD
- **Sistema Operativo**: Windows 10

Debido a las restricciones impuestas por el gobierno para evitar el avance de la pandemia, el proyecto y los materiales fueron trasladados, por lo que a partir del tramo final de desarrollo este fue ejecutado en un equipo con menor capacidad:

- **RAM**: 8GB
- **CPU**: Intel i5-4460 4 CPU $\sim$ 3.2GHz
- **Gráfica**: NVIDIA GeForce GTX 960 (VRAM: 4GB)
- **Almacenamiento**: 250GB SSD
- **Sistema Operativo**: Windows 10

4.1.1. Impresión 3D del adaptador para Leap Motion

Durante las etapas iniciales del desarrollo, que consistió en investigar las tecnologías, descubrimos que era necesario un soporte para que **Leap Motion** se sujetara en la parte frontal del casco **Oculus Rift**. Este adaptador se muestra en la fig.19 y cuesta alrededor de 30€.

El problema es que hay que iniciar una serie de trámites para que el laboratorio pueda adquirir cualquier material, por lo que comprar este adaptador supondría un gran retraso de tiempo. Sin embargo, se nos ocurrió utilizar una impresora 3D para crear la pieza. La impresora del SIC es un modelo **Ultimaker 3** como podemos ver en la fig.24 y el diseño de la pieza fue sacado de la web **Thingiverse** proveniente de la cuenta oficial de **Leap Motion**[24], es decir, que la propia compañía dejaba el modelo 3D a disposición de las personas que quisieran imprimirlo por sí mismas. El modelo impreso tiene una curvatura en la base de apoyo, ya que la parte frontal de nuestro casco VR (y de algunos otros también) es ligeramente curvada.

Figura 24: Impresora 3D Ultimaker 3

Fuente: https://commons.wikimedia.org/wiki/File:Ultimaker_3_between_Ultimaker_2_Extended%2B_and_Ultimaker_2%2B.jpg

Por último, una vez impresa la pieza, se utilizó cinta de doble cara debido a su gran fuerza de sujeción (otras cintas no soportaban el peso de **Leap Motion**). El resultado final es el que se muestra en la fig.25.

Figura 25: Resultado final del adaptador VR de Leap Motion

4.2. Unity

En el inicio del proyecto poseíamos pocos conocimientos de **Unity** y nunca habíamos trabajado con **Leap Motion** ni **Oculus Rift**, por lo que el primer objetivo fue aprender a crear un entorno donde funcionaran ambos. A partir de eso, sería necesario **crear un fondo** visualmente agradable y una **serie de objetos** con los que interactuar. Esta sección pretende profundizar cada uno de estos procedimientos.

4.2.1. Integración entre Unity, Oculus Rift y Leap Motion

Empezamos por investigar a **Oculus Rift** y decidimos basarnos en un tutorial de *Youtube*[25] para aprender a utilizar los *assets* de **Oculus** en **Unity**. El tutorial te muestra cómo crear una especie de *shooter* (género de videojuegos en primera persona donde el jugador suele disparar armas). Para ello, se creaba una pistola y los mandos *Oculus Touch* servían para disparar con los gatillos inferiores. Por tanto, el entorno consistía en una mesa donde se encontraban diversos cubos a los que podíamos disparar para así poner a prueba nuestra puntería.

Tras ver la manera en la que se comportaban los scripts y componentes necesarios de **Oculus**, decidimos pasar a **Leap Motion**. Para ello, desde el entorno que ya teníamos, eliminamos el componente que controlaba los mandos *Oculus Touch* y los diversos elementos del *shooter*.

A partir de aquí, introdujimos los *assets* de **Leap Motion**[26] y pudimos compro-

bar que a raíz de los ejemplos que traen estos se puede extraer un componente que controla nuestra manos y está preparado para VR. Por último, colocamos objetos provenientes también de los ejemplos, que tienen los *scripts* necesarios para la interacción con nuestras manos y nos queda un entorno (sin decoración alrededor) con una mesa donde se encuentra el usuario y 3 objetos; todo esto pudiendo ser utilizado con ambos **Leap Motion** y **Oculus Rift**.

Cabe mencionar que la **versión de Unity** que utilizamos durante todo el desarrollo del proyecto es la **2018.4.17f** debido a que fue la que mejor compatibilidad mostraba con los *assets* de Oculus y Leap Motion y nuestra licencia es **Unity Personal**.

4.2.2. Creación del fondo

Ya que teníamos una base que funcionaba con las tecnologías mencionadas, era hora de ir construyendo algo visualmente agradable y más serio. Por ello, recurriendo a un *asset gratuito*[27] de la **Unity Store**, construimos el fondo mostrado en la fig.26.

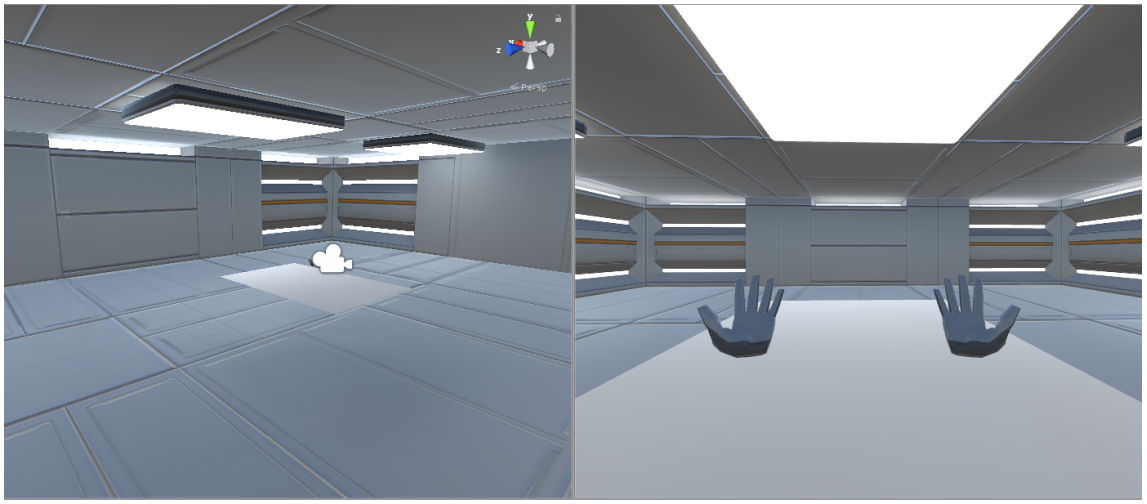

Figura 26: Fondo creado inicialmente

Este *asset* provee distintos elementos para poder construir un decorado *scifi* o futurista. El ejemplo que trae nos muestra una especie de nave espacial con distintos pasillos y puertas. A raíz de este, adaptamos un fondo que estuviera decorado de igual manera pero siendo una habitación simplemente, de forma que nuestro entorno es un amplio habitáculo cuadrado con una mesa en medio donde se encuentra el usuario y los 3 objetos para interactuar mencionados anteriormente.

Esta fue la primera versión que mostramos en el laboratorio a varias personas ya que demostraba de manera simple pero eficaz el potencial que podría llegar a tener el entorno en un futuro.

Cabe mencionar que en las etapas finales del desarrollo decidimos añadir otros dos fondos que fueran más acorde a un ámbito laboral o empresarial, para que las simulaciones tuvieran una mayor verosimilitud.

El primero de ellos lo podemos ver en la fig.27 y está compuesto de varios *modelos gratuitos de Internet*[28][29][30], este fondo fue formado a partir del modelo de un **almacén** y dentro de este fuimos colocando diversos elementos industriales: **barriles de aceite, containers y bombonas de materiales inflamables**. Aunque la imagen sólo muestra la vista delantera, la parte trasera está decorada también.

Figura 27: Segundo fondo añadido

El segundo se aprecia en la fig.28 y está compuesto por alguno de los modelos anteriores y otros nuevos[31][32]. Este fondo fue creado a raíz de un conjunto de edificios, se cerró para que diera la sensación (de nuevo) de un habitáculo o zona cuadrada, se decoró con alguno de los modelos del fondo anterior y además se añadió una serie de **herramientas manejables**, para poder dar algo más de realismo a este fondo.

4.2.3. Desarollo de los objetos

Una vez acabado el fondo, debíamos construir una serie de objetos propios de nuestra aplicación, para ello quisimos buscar algo simple y que permitiera interacciones eficientes. Es decir, en vez de construir un sistema complejo que nos llevara mucho tiempo diseñar y modelar, decidimos buscar algo más **práctico**. Es por ello que optamos por crear piezas del **Tetris**.

Para ello, utilizamos un cubo base de Unity al cuál le dotamos de distintos *sprites* con temática similar al *Tetris*. Cabe mencionar que este *cubo origen* posee un componente *Collider*, los cuáles permiten que la pieza final posea colisiones exactamente

Figura 28: Tercer fondo añadido

en toda su forma. Por último, uniendo varios de estos formamos las piezas mostradas en la fig.29.

A cada una de las piezas se le colocará la etiqueta (*tag*) correspondiente a su color, de manera que podamos conocer el tipo de pieza desde el código.

Una vez creada cada una de las piezas, debemos añadirle los correspondientes *scripts* y componentes que nos permitan interactuar con ellas. Para ello, le añadimos un **Rigidbody**, componente nativo de Unity que dota al objeto de gravedad y colisiones (el objeto no atravesará a otros objetos) y un *script* **InteractionBehaviour**, componente proveniente del *asset* de **Leap Motion** que nos permite realizar distintas acciones con el objeto que ha recibido el *script* (agarrar, levantar, empujar...). Por último, mencionar que a cada una de las piezas se le añade un objeto que muestra su ID, el cuál será el número de piezas creadas hasta ese momento (véase fig.30); para así referenciar a cada una de ellas de manera inequívoca.

Tras esto, tenemos la base sobre la que construir interacciones, simulaciones y por supuesto, las funciones con Alexa.

Figura 29: Objetos creados para el entorno

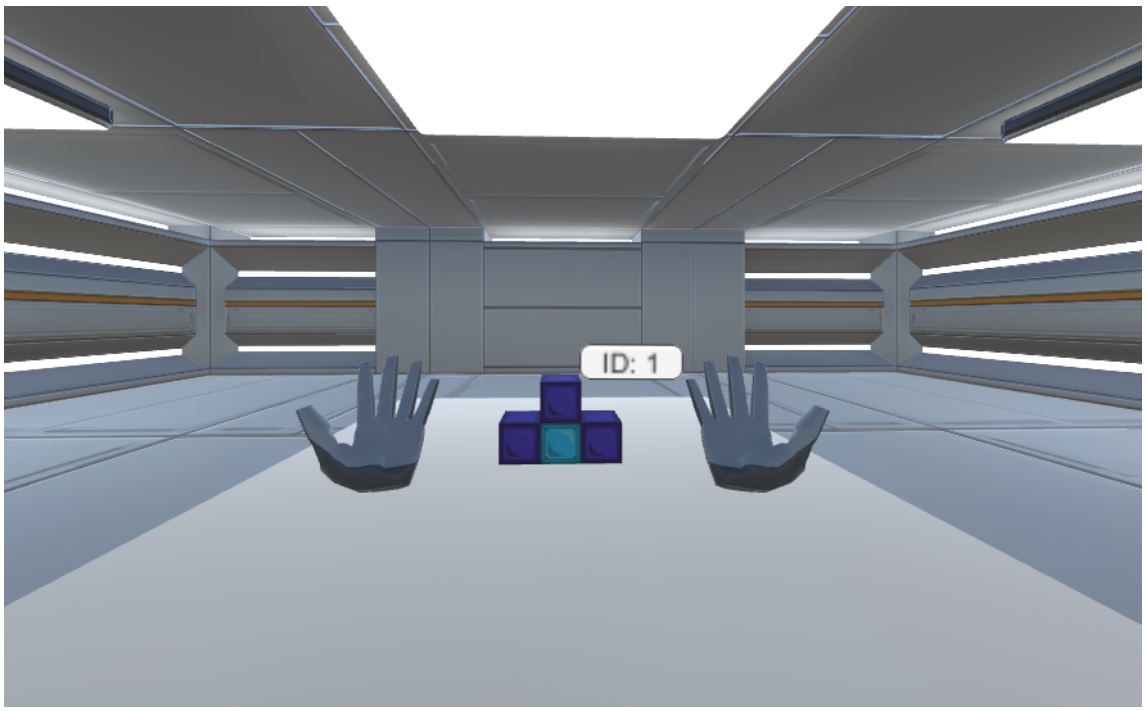

Figura 30: Objeto con su etiqueta indicando ID

4.3. CloudAMQP

Comenzando la parte de conectar a Unity y Alexa, debíamos buscar algún sistema que permitiera una comunicación entre ambos. Uno de los requisitos principales que buscábamos es que el programa dispuesto a recibir los mensajes no consumiera demasiados recursos, ya que esto podría producir la ralentización de nuestro entorno.

Una de las ideas iniciales fue investigar acerca de algún sistema de API REST. Sin embargo, el **elemento servidor** (el que recibe el mensaje) debe ejecutar un programa con ciclos infinitos para así recibir el mensaje correctamente. Por tanto, consume muchos recursos y no cumple el requisito anteriormente mencionado.

A raíz de esto, descubrimos el **modelo de cola de mensajes**, el cuál permite ejecutar al **elemento servidor** sin ciclos infinitos y su forma de entregar/recoger mensajes es a través de una **cola**. La tecnología es la adecuada para nuestro proyecto y ahora quedaba la tarea de escoger una API que tuviera soporte en la nube.

Es por ello que decidimos usar **CloudAMQP** (basado en RabbitMQ) ya que ofrece las siguientes ventajas en nuestro proyecto:

- **Evita sobrecarga en el entorno**. Al tratarse de un modelo de *cola de mensajes* evitamos consumir recursos continuamente. Por tanto, la fluidez de nuestro entorno no se verá afectado al incluir esta tecnología.

- **API disponible en C# y *Javascript***. *CloudAMQP* y *RabbitMQ* poseen una API para **C#**[33] y **Javascript**[34], por lo que la integración de Unity y Alexa será liviana y rápida.
- **Ofrece un plan gratuito**. *CloudAMQP* posee un plan gratuito limitado a un número de mensajes enviados al mes. Hemos usado este plan en el desarrollo de este proyecto para así evitar costes.

Dado que hemos elegido ya un servicio en la nube, pasamos a configurarlo. Primero creamos una cuenta en **CloudAMQP** con licencia *Little Lemur*, ya que es el plan gratuito con peticiones limitadas al mes. A continuación, dentro de **RabbitMQ Manager** (en la propia web) creamos la cola *piezas* y seleccionamos la propiedad **durable** a **true**, para que los mensajes se queden almacenados en la cola hasta que una entidad los reciba, es decir, no perderemos los mensajes aunque pase un tiempo hasta su recogida. Tras esto, desde el panel principal de nuestro perfil en **CloudAMQP**, copiamos la dirección **AMQP URL** y la insertamos en nuestro código (tanto si es Javascript para el caso de Alexa o C# para el caso de Unity) y a continuación especificaremos la cola con la que queremos conectar, que en nuestro caso es *piezas*; a partir de aquí ya habremos establecido conexión con el servicio en la nube y estaremos preparados para recibir los mensajes desde nuestra aplicación en Unity.

Una vez integrado **CloudAMQP** en nuestro entorno, decidimos crear un sistema para que, dependiendo del mensaje recibido, Unity realizara una acción u otra. Para ello, creamos un *script* en Unity el cuál al inicio de la aplicación comenzaba la conexión con la **cola en la nube**. Tras esto, si recibía un mensaje, cambiaba el valor de una variable local alojada en el *script* por el contenido del mensaje recibido. Mientras, cada *frame del entorno* íbamos comprobando si esa variable había cambiado. Si esto ocurría, dependiendo del valor que tomara, realizaríamos una acción u otra. Tras realizar la acción, devolvíamos la variable local a su valor predeterminado. Este era el proceso que seguíamos para cada mensaje.

Durante las pruebas donde desarrollamos este sistema, enviábamos los mensajes mediante un programa escrito en *Javascript* y creado precisamente para comprobar si las acciones se realizaban adecuadamente. Sin embargo, el objetivo ahora es que este programa que envía los mensajes sea ejecutado por Alexa.

4.4. Alexa

Una vez que estuvo listo la manera de hacer que Unity realizara acciones a través de mensajes, necesitábamos que Alexa pudiera enviarlos. Para ello, descubrimos que lo necesario era **crear una skill propia** para el proyecto y en cada una de las conversaciones con Alexa **colocaríamos nosotros el código oportuno**. En esta sección vamos a tratar ambos aspectos.

4.4.1. Creación de la skill

A la hora de organizar nuestra skill debíamos crear un *modelo de diálogo* de forma que en cada una de las conversaciones con Alexa se introdujera una acción a realizar en Unity y además estas acciones se realizaran según la información recibida del usuario.

Para ello, tenemos que tener en cuenta varios conceptos derivados de las *skills*:

- **Invocation Name**: el nombre con el que se iniciará nuestra *skill* y podremos a continuación indicarle acciones. En nuestro caso elegimos **Objetos VR**.
- **Intent**: define la acción a realizar por la *skill*. Con este elemento seleccionaremos el tipo de mensaje a enviar a Unity.
- **Utterances**: las distintas formas que encontramos para invocar una acción: "*crea una pieza*", "*genera una pieza*," "*pon una pieza*"...
- **Slot**: corresponden a diferentes valores introducidos por el usuario. Estos nos servirán para saber específicamente lo que quiere el usuario. Pueden ser predefinidos por Amazon o podemos crear los nuestros.

Como podemos ver en la fig.31 tenemos un ejemplo de conversación donde el usuario pide el horóscopo. En este caso, el *intent* será **GetHoroscope** y la forma de invocarlo (*Utterance*) es "give me the horoscope". Por último, el valor del signo del zodíaco será almacenado en el *slot* **Sign** el cuál desde el código podremos referenciar para saber el signo seleccionado y dar una respuesta concisa.

Figura 31: Ejemplo de intent y slot en una *skill*
Fuente: https://www.screenmedia.co.uk/lab/alexa-skill-development-understanding-intents-utterances-and-slots/

Entre los intents que el usuario creará debe rellenar 4 obligatorios: **AMAZON CancelIntent** (para cancelar una acción), **AMAZON HelpIntent** (para obtener ayuda sobre el uso de la *skill*), **AMAZON StopIntent** (para cerrar la *skill*) y **AMAZON NavigateHomeIntent** (lleva al usuario a la pantalla principal del dispositivo y cierra la skill).[35]

Por tanto, nuestra skill se invoca mediante: "*Alexa, abre Objetos V. R.*", "*Alexa, inicia Objetos V.R.*" o similares. A raíz de aquí se nos devolverá el siguiente mensaje de bienvenida: "*Bienvenido a Objetos V. R. Especifique una acción a hacer a continuación.*".

Desde este momento hemos entrado en la skill y podremos realizar varias acciones:

- **Invocar alguna operación implementada**. Podemos invocar algunas de las funciones implementadas que modificarán nuestro entorno en Unity. Todas las funciones finalmente disponibles serán mencionadas más adelante.
- **Cerrar la skill**. Si queremos cerrar la skill tendremos que indicar "*Alexa, para*", "*Alexa, cierra*" o "*Alexa, quita*" y entonces se nos comunicará el mensaje: "*Se ha cerrado la skill*".
- **Ayuda**. Mediante "*Alexa, ayuda*", "*Alexa, help*" o "*Alexa, qué puedo hacer*" invocaremos el *intent* de ayuda y se nos comunicará lo siguiente: "*Puedes utilizar los comandos Crear Pieza, Cambiar Entorno, Crea Objeto, Genera Objeto, Parar Simulación o los correspondientes a iniciar una simulación*".

Para esta *skill* hemos utilizado 4 slots: **AMAZON Number** que posee una biblioteca de números, **Color**: creado por nosotros y que nos permite controlar que el color especificado esté relacionado con alguna de las piezas que poseemos, **Objetos:** con una serie de objetos permitidos por nosotros y **Orden**: también creado por nosotros para indicar orden de *primero* o *segundo*.

Por último, cabe mencionar un par de aspectos, el primero es que el diseño de la *skill* siguió el estándar de **Certificación de *skill***[36] e incluso lo aprobó; de forma que nuestra *skill* pudo ser publicada para todos los usuarios, pero por cuestiones del proyecto se decidió mantener en *etapa de desarrollo*. El segundo aspecto es que para probar la *skill* se ha utilizado un dispositivo **Amazon Echo Dot** personal (véase fig.32), ya que aunque la *skill* no esté pública, debido a que la cuenta vinculada a este y la de desarrollo son la misma, podemos utilizarla directamente desde el dispositivo.

4.4.2. AWS Lambda

Una vez teníamos modelada nuestra *skill*, era hora de colocar el código (de *CloudAMQP*) correspondiente en cada uno de los *intents*. Nuestro código para enviar mensajes a la nube estaba escrito en *Javascript*, el mismo lenguaje que utiliza Alexa, sin embargo, usamos una librería no nativa llamada **amqplib** [37] (siguiendo las pautas de la documentación de *CloudAMQP*) y Alexa sólo soporta las librerías nativas de *Node.JS* por lo que teníamos que buscar otra manera de introducir esta librería en nuestra *skill*.

Figura 32: Dispositivo Echo Dot usado para el proyecto

Investigando descubrimos que **AWS Lambda** admite *Javascript* y bibliotecas de cualquier tipo, de forma que podemos incluir una carpeta que posea la librería disponible y el código que la usa para mandar el mensaje a la **cola en la nube**. Para ello, creamos una nueva función en **AWS Lambda** y siguiendo un tutorial que nos permite crear una **API REST entre Alexa y AWS Lambda**[38] utilizamos **Serverless**[39] (un *framework* de código abierto para *Node.JS* que permite subir el código a *AWS Lambda* y ponerlo en funcionamiento) para crear un proyecto nuevo de Node.JS en el cuál colocaríamos los archivos necesarios. Para ello ejecutamos el comando **serverless create**.

En este proyecto se encuentran 3 archivos: **.npmignore**, **handler.js** y **serverless.yml**. El primero sirve para indicar a Node.JS los archivos que no pertencen a los paquetes (librerías) del proyecto; el segundo para indicar los aspectos relevantes de nuestra función en **AWS Lambda** (permisos, roles, ID...) y el tercero para controlar los aspectos referidos a Serverless. En este último colocaremos información sobre el proveedor en la nube, las funciones que incluiremos en Lambda y los recursos que utilizan estas funciones. Para la parte de las funciones creamos *send(mensaje)*, siendo *mensaje* una cadena de texto que nos permite enviar la acción a realizar a **CloudAMQP** mediante **amqplib**. A continuación, con cada archivo correctamente rellenado, ejecutamos el comando **sls deploy** y estos son subidos a nuestra función **AWS Lambda**.

A partir de aquí, nuestro objetivo es que Alexa pueda invocar de alguna forma esa función de **AWS Lambda**. Para ello, colocamos el componente **API Gateway** en Lambda, ya que este nos permite disponer de operaciones de REST API, por

Figura 33: Esquema del recorrido que toma el mensaje que queremos enviar

lo que seleccionamos el método POST. Debido a que las peticiones de *HTTP* se incluyen como librería nativa de *Node.JS*, Alexa podrá realizar un POST a la dirección especificada por el componente **API Gateway** y entonces activar la función *send(mensaje)*.

Por tanto, como se muestra en el esquema de la fig.33, Alexa generará el mensaje a partir de la conversación que haya tenido con el usuario (*intents* y *slots*) y lo enviará mediante la petición POST a **Lambda**. Entonces se ejecutará la función *send* que colocará el mensaje en la cola de **CloudAMQP** mediante la librería **amqplib** y a continuación Unity lo recogerá y realizará la acción en el entorno.

4.4.3. Funciones

Una vez realizado el sistema que permite comunicar a Unity y Alexa, era hora de definir las acciones que será capaz de realizar y que por tanto afecten al entorno.

En esta sección vamos a tratar todas las funciones que se incorporaron finalmente a este proyecto. Cada palabra que vaya encerrada en {} indicará un *slot*, cuyo valor será especificado por el usuario en el momento de invocar la acción.

- **Crear Pieza**. Crea una pieza de un cierto color en el entorno. La invocación se realiza de la siguiente forma: "*Alexa, abre Objetos V. R. y crea/genera/pon una pieza de* {*Color*}". El *slot* **Color** permitirá los colores que hacen característica a cada una de las piezas: *morado, amarillo, verde, rojo, azul y naranja.* El mensaje que se enviará por tanto será: "crea{Color}".

- **Eliminar Pieza**. Elimina una pieza según el ID de esta. La invocación se realiza de la siguiente forma: "*Alexa, abre Objetos V. R. y elimina/quita la pieza (con ID)* {*AMAZON Number*}". El *slot* **AMAZON Number** nos indicará el número (ID) de la pieza. El mensaje enviado será "{ID}".

- **Cambiar Entorno**. Cambia el fondo del entorno de forma cíclica, empezando por el 1 hasta el 3 (véase fig.26, 27 y 28). Para invocarlo deberemos indicar

lo siguiente: "*Alexa, abre Objetos V. R. y cambia/modifica el entorno*". El mensaje enviado es el siguiente: "*entorno*".

- **TicTacToe**. Comienza la simulación del *tictactoe*. Para invocarlo tendremos que decir lo siguiente: "*Alexa, abre Objetos V. R. y abre/comienza simulación de tictactoe/tres en raya* {*Orden*}". El *slot* **Orden** nos permitirá definir el turno que va a tomar el usuario, por lo que los valores posibles son *primero* o *segundo*. El mensaje enviado será: "*tictactoe*{*Orden*}".

- **Simulación Guiada**. Comienza una simulación guiada. Para invocarlo deberemos indicar lo siguiente: "*Alexa, abre Objetos V. R. y abre/comienza/inicia simulación guiada*". El mensaje enviado será: "*guiada*".

- **Simulación Libre**. Comienza una simulación no guiada. Para invocarlo deberemos decir: "*Alexa, abre Objetos V. R. y abre/comienza/inicia simulación libre*". El mensaje enviado será: "*libre*".

- **Parar Simulación**. Elimina el tablero y todos los elementos colocados en él, terminando la simulación actual. Para invocarlo debemos decirle a Alexa: "*Alexa, abre Objetos V. R. y elimina el tablero / cierra/para la simulación*". El mensaje enviado será: "*eliminar*".

- **Crear Objetos**. Busca un objeto en Internet y lo añade al entorno. Para invocarlo deberemos indicar lo siguiente: "*Alexa, abre Objetos V. R. y busca/genera un objeto* {*Objeto*}". El *slot* **Objeto** tiene definido 10 objetos disponibles a buscar, aunque se ha modelado para poder introducir cualquier palabra, ya que no se ha colocado una validación que acepte únicamente las palabras indicadas. El mensaje enviado será: "*genera*{*Objeto*}".

4.5. Simulaciones

A estas alturas nuestro entorno posee diversos **objetos interactivos**, **varios fondos seleccionables** y **acciones mediante Alexa**. Por tanto, nuestra nueva meta es implementar simulaciones con distintas formas de interactuar.

Para esto nuestra idea fue crear un tablero con celdas que nos permitiera colocar las piezas y a raíz de ahí poder construir las simulaciones bajo una serie de reglas. Esta sección tratará ambos aspectos: **la creación y definición del tablero** y **el diseño de las simulaciones**.

4.5.1. Creación del tablero

La idea principal del tablero era encontrar un sistema para que el jugador colocara las piezas sin tener que recurrir a una precisión exacta. Es decir, según la posición en que pongamos la pieza en el tablero, esta se "magnetizará".

Buscando tutoriales dimos con *Unity 3d Place Object On Grid*[40] que muestra la forma de realizar una aplicación que crea un tablero de m x n cuadrados (podemos elegir las dimensiones) y cuando clicamos en una de las casillas se genera un cubo que queda colocado en este.

A raíz de la idea principal que conlleva esto, portamos la aplicación a nuestro entorno y desde ahí fuimos trabajando para crear el tablero que deseamos. Primero fuimos adaptando la aplicación para que funcionara con una de nuestras piezas (elegimos la morada por orden) ya que la aplicación simplemente sirve para un cubo y más tarde lo extrapolamos al resto de piezas.

Esto funcionaba colocando las piezas en un cierto orden y orientación, sin embargo, a la hora de probarlo agarrando con las manos y colocando las piezas de otra determinada forma, la aplicación funcionaba de manera errónea. El problema a solucionar es que debíamos detectar qué pieza se colocaba y de qué forma y a raíz de ahí marcar las casillas correspondientes como *ocupadas*. Para llevar a cabo esto, utilizamos varias técnicas que explicamos a continuación.

Para el tablero creamos una estructura de datos llamada **Square** que iría asignada a cada una de las casillas. Esta estructura contiene diferentes variables:

- **Bool isPlaceable**. Nos indicará si alguna pieza está ocupando esa casilla en este momento.
- **Vector3 cellPosition**. La posición de la casilla en el entorno (instancia de Unity).
- **Transform obj**. Una referencia del objeto colocado.
- **Int i/j**. Estos enteros nos permiten saber la posición de la casilla en el tablero (i corresponde a filas y j a columnas).
- **Square up/left/right/down**. Una referencia a las casillas superior, inferior, derecha e izquierda de la casilla actual. Nos servirá para disminuir el tiempo de búsqueda de casillas.

Al inicio de la creación del tablero hicimos que cada una de las casillas rellenara su variable *Square* y estas se almacenaban en una matriz de m x n, para una vez que vayamos a colocar objetos tengamos todas las casillas correctamente preparadas.

Lo siguiente a realizar es cómo detectar que una pieza ha entrado en contacto con el tablero. Para ello, ya que cada pieza tiene su componente *Rigidbody* podemos detectar la colisión con otros objetos. En este caso a cada casilla se le colocó la etiqueta *Cell*, entonces creamos el *script* **DetectCollision** que implementa la función **OnCollisionEnter** la cuál se activa cada vez que el objeto entra en contacto con

otro. En este momento simplemente tendremos que comprobar que es una casilla (su etiqueta es igual a *Cell*) y activar la función que coloque la pieza.

Cabe mencionar que en el inicio colocábamos las piezas según el cubo que había colisionado con el tablero, es decir, colocábamos la pieza tomando como referencia la posición en la que había chocado. Esto podía producir movimientos indeseados ya que hay que tener mucha precisión para hacer que la pieza choque únicamente por el lado que deseamos y así se coloque de la manera adecuada. Como solución a esto, decidimos que la posición de cada pieza coincidiría con un **cubo central** propio de cada una, el cuál marcará la **posición que tomará de referencia la pieza** al colocarse en el tablero y además decidimos que el sistema siempre tomará como referencia esta posición y no la zona donde se ha producido la colisión, por lo que necesitamos proporcionar al usuario algún tipo de estímulo para que comprenda qué cubo es el central. Es por ello que cada pieza tiene un cubo de un color distinto (**azul turquesa**) como vemos en la fig.29, este servirá para que el usuario lo tome como guía o referencia a la hora de colocar la pieza en el tablero.

Una vez que se ha detectado la colisión se activará la función **PlaceObjectOnGrid**, en esta se buscará la casilla que se encuentre más cercana a la posición de la pieza (cubo turquesa) y a partir de ahí, se detectará **la rotación** y **tipo de pieza** y se comprobará que las distintas casillas contiguas estén disponibles (y no sobresalgan del tablero) antes de colocar la pieza. Para detectar el **tipo de pieza** simplemente tenemos que comprobar la etiqueta que posee el objeto y para la **rotación**, construimos la función *Nearest*, la cuál a partir del valor de la rotación (en este caso del eje Y) devolverá el grado que esté más cercano: 0, 90, 180 o 270. Estos grados son las posibles rotaciones que puede poseer la pieza al colocarla y cada uno de ellos necesitará una comprobación distinta de las casillas disponibles. En la fig.34 encontramos las rotaciones de la pieza morada; en el primer caso (0^o) habrá que comprobar que las casillas derecha, izquierda y superior están libres (*isPlaceable* verdadero) mientras que en el segundo caso (90^o) habrá que comprobar que la casilla derecha, inferior y superior estén libres; por supuesto, la casilla donde se encuentra el cubo turquesa también se comprobará. Por tanto, creamos distintas funciones nombradas con el color de cada pieza, las cuáles mediante la rotación (corregida por la función *Nearest*) comprueban cada una de las casillas correspondientes. Tenemos que añadir que **PlaceObjectOnGrid** devolverá una referencia *Square* con el cuadrado donde se ha colocado el objeto; si no se ha colocado, devolverá *null*.

Si las casillas correspondientes están disponibles, las funciones colocarán la pieza. Para ello, se modifica la posición de la pieza (coincide con el cubo turquesa) para que sea igual a la casilla más cercana y la rotación corresponderá con lo devuelto por *Nearest*. A continuación colocamos el componente *Rigidbody* con *detectCollisions* a *false* ya que ignora las colisiones para así facilitar la colocación de las siguientes piezas; también desactivamos la capacidad de interactuar con ella a través de *Leap*

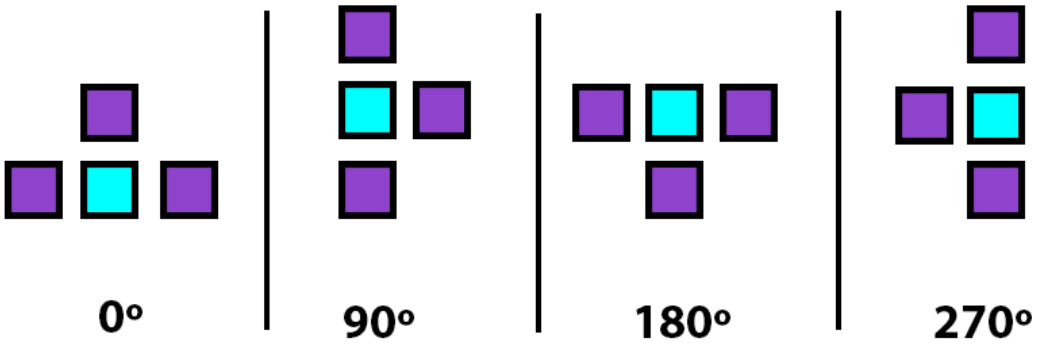

Figura 34: Rotación de la pieza morada

Motion (*InteractionBehaviour – ignoreContact = true*) y por último cambiamos la etiqueta de la pieza a *Colocada*. Cabe mencionar además que creamos un vector llamado **piezasColocadas** el cuál tiene el *tipo de dato* **Pieza**, que nos informa del **color de pieza**, su **rotación** e incluso qué casilla ocupa en el tablero. Este vector es actualizado cada vez que se coloca una pieza y nos sirve para futuras operaciones.

Desde este momento, el tablero funcionaba tal y cómo queríamos ya que daba la sensación de magnetizar las piezas a este. Sin embargo, en distintas pruebas descubrimos un pequeño comportamiento que no habíamos tenido en cuenta y que sacaba al usuario de la experiencia. El comportamiento es el siguiente: todo lo anterior estaba pensado para que la pieza cayera hacia delante en la mesa (eso supone la rotación del eje Z a -90º) pero no habíamos tenido en cuenta que cayera hacia el otro lado (eje Z a 90º). Esto se traducía en que si la pieza giraba mucho en la mano del jugador, a la hora de colocarla esta tomaba una posición que no era la que le correspondía. Para ello, descubrimos que cada pieza con su eje Z a -90º tenía una "*traducción*" similar cuando el eje Z tomaba 90º.

Es por ello que creamos la función **corregirAngulo**, que tomaba la rotación del eje Y y del Z devuelta por *Nearest* (ángulos más cercanos) y el tipo (etiqueta) de pieza y devolvía el grado "*traducido*" para este objeto, de forma que en el eje Y iría el "*traducido*" y en el Z el devuelto por *Nearest*. Con esto logramos una solución eficaz ante el último problema del tablero.

Ya que habíamos conseguido un sistema funcional para **colocar** las piezas en el tablero, decidimos crear otro para que se pudieran **retirar** las piezas. En un primer momento pensamos en la posibilidad de extraerla con nuestras propias manos.

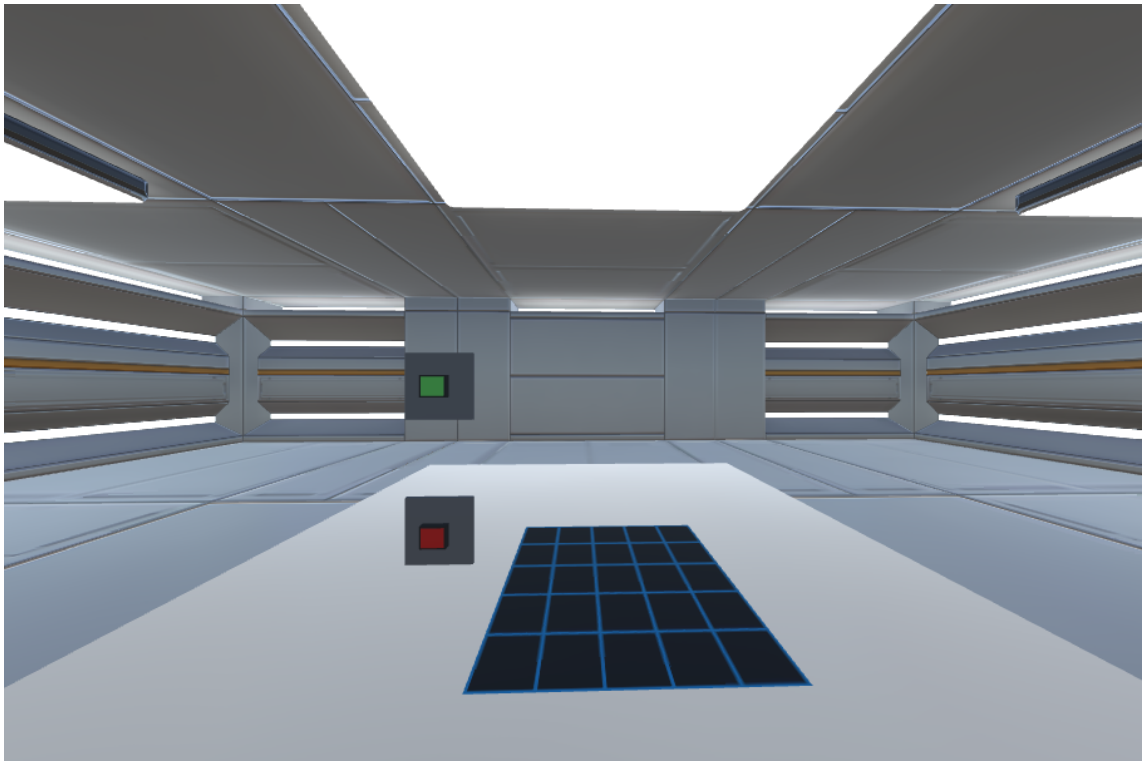

Figura 35: Tablero y botones del entorno

Para ello, simplemente tendríamos que comprobar si la pieza había sido agarrada (booleano *isGrasped* en componente de *Leap Motion*). Sin embargo, tras varios días probando el funcionamiento, descubrimos que de alguna forma agarrar una pieza crea conflicto con detectar colisiones de la misma, por lo que la pieza no se podía retirar y si lo hacía, esta no poseía alguna de las características anteriores (como la gravedad). Por tanto, una solución implementada para las simulaciones fue añadir un **botón rojo** que permitiera al usuario deshacer su movimiento. Además, este también es útil si nuestra pieza está lejos y nuestro entorno físico no nos permite acercarnos a ella. El funcionamiento del botón es el siguiente: si pulsas el botón, la pieza se elimina de donde esté y reaparece justo delante del usuario. Para ello, dentro del *script* **PlaceObjectOnGrid** colocamos la función *RemoveFromGrid* la cuál recibiendo la casilla en la que se encuentra la pieza hará el paso inverso a la función de **colocar**: revisará en el vector **piezasColocadas** cuál es la pieza a eliminar y según el tipo y rotación pasará a *liberar* todas las casillas que había ocupado según su forma; finalmente eliminará el objeto del entorno y lo creará de nuevo en la posición inicial.

Con esto nos surge una nueva duda, **cuándo detectar que el movimiento del usuario es el definitivo**, es decir, la posición actual de la pieza no es una errata. Para ello, igualando la solución anterior, dispusimos de un botón verde el cuál será

pulsado cuando la respuesta del usuario sea definitiva y entonces pasaremos a la siguiente fase de la simulación, en la fig.35 podemos ver los botones y el tablero colocado en el entorno. Tenemos que mencionar que estos botones forman parte de los *assets* de **Leap Motion** con lo que serán pulsados de manera sencilla con nuestras propias manos. Para añadir algo de realismo, al pulsarlos se escucha un sonido de *clic*.
Por último, a continuación vamos a comentar un par de aspectos a tener en cuenta:

- Debido a que cada pieza cambia su etiqueta a "*Colocada*" una vez colocada en el tablero, esto nos permite de manera muy sencilla realizar la función **Eliminar Tablero** que vimos en la sección 4.4.3 ya que simplemente tendremos que eliminar todos los objetos con etiqueta "*Colocada*" y "*Cell*". Para detener las simulaciones usamos otra serie de variables auxiliares.

- Como consecuencia de los conflictos entre *Rigidbody* y la sujección de objetos mediante **Leap Motion** comentados anteriormente, la única manera de asegurar la correcta colocación de las piezas es dejarlas caer desde una pequeña altura en la posición que deseamos, ya que si ponemos la pieza directamente en las casillas (tocando el tablero) y después la soltamos, la colisión no se detectará de manera correcta y por tanto no llamará a **PlaceObjectOnGrid** para "magnetizar" y corregir la posición de la pieza.

4.5.2. TicTacToe

La primera simulación implementada fue el **TicTacToe** o juego del tres en raya. Nuestra decisión de incluirlo viene a la hora de querer introducir una simulación que permita una cierta interacción con una **Inteligencia Artificial**. Para esto hemos utilizado el método **Minimax**.

Antes de pasar a explicar el funcionamiento de la simulación, vamos a explicar cómo hemos realizado **Minimax**. Primero, nos hemos guiado del esquema explicado en la sección 3.7.1 y realizamos una función que posee ciclos infinitos a la espera de que el juego llegue a un final. Esta primera función correspondería a la función *MINIMAX-DECISION* de la fig.23.

A continuación hemos pasado a crear las funciones *valorMin* y *valorMax*, las cuáles corresponden en el pseudocódigo de la fig.23 con *MIN-VALUE* y *MAX-VALUE*. Por supuesto hemos creado una serie de funciones auxiliares: **terminal** (comprueba si se ha llegado a un estado final), **aplicaJugada** (a partir de cierto nodo o estado añade la jugada indicada) y **esValida** (devuelve verdadero si la jugada indicada es válida en el estado dado).

Utilizando lo anterior conseguimos implementar una IA que nunca pierde al *TicTacToe*. Sin embargo, el input sigue siendo a través de la terminal, por lo que debemos

adaptar el código para que se permita a la IA crear su jugada en el entorno y que además espere a que el jugador coloque su pieza.

Primero debíamos crear algún objeto o pieza que simulara al *círculo* y la *cruz* del juego. Para ello, usando un cubo (con la altura reducida) y añadiéndole un *sprite* gratuito, conseguimos lo mostrado en la fig.36. A continuación debíamos hacer que nuestra IA, una vez seleccionado su movimiento, creara a raíz del prefab *cruz* un objeto en la posición correspondiente y a una cierta altura del tablero, para que este caiga por la gravedad justamente en la posición que eligió. Para ello, como nuestra IA devuelve un único número que corresponde a la casilla elegida (entre 0 y 8 ya que el tablero tiene 9), debíamos obtener los índices correspondientes de esa casilla en la matriz que almacena todos los *Square*, por lo que utilizamos la operación mostrada en la fig.37; el índice i corresponderá a la fila y j a la columna. Una vez que tenemos el *Square* correspondiente a la casilla elegida simplemente debemos instanciar el prefab *cruz* en esa posición (variable **cellPosition** de *Square*) a una altura que nos parezca adecuada, en este caso fue a *0.2* del tablero. Cabe añadir que hubo que extender la función **PlaceObjectOnGrid** para que aceptara también la colocación de los *prefab circulo* y *cruz*, por lo que el funcionamiento es idéntico (incluso más sencillo) que lo explicado en la sección 4.5.1; además, esto nos permitirá colocar la pieza únicamente en celdas que no estén ya ocupadas. Por último, añadir que se colocó un pequeño sonido de alerta cuando la IA coloca su pieza, para avisar de esta manera al usuario de que es su turno.

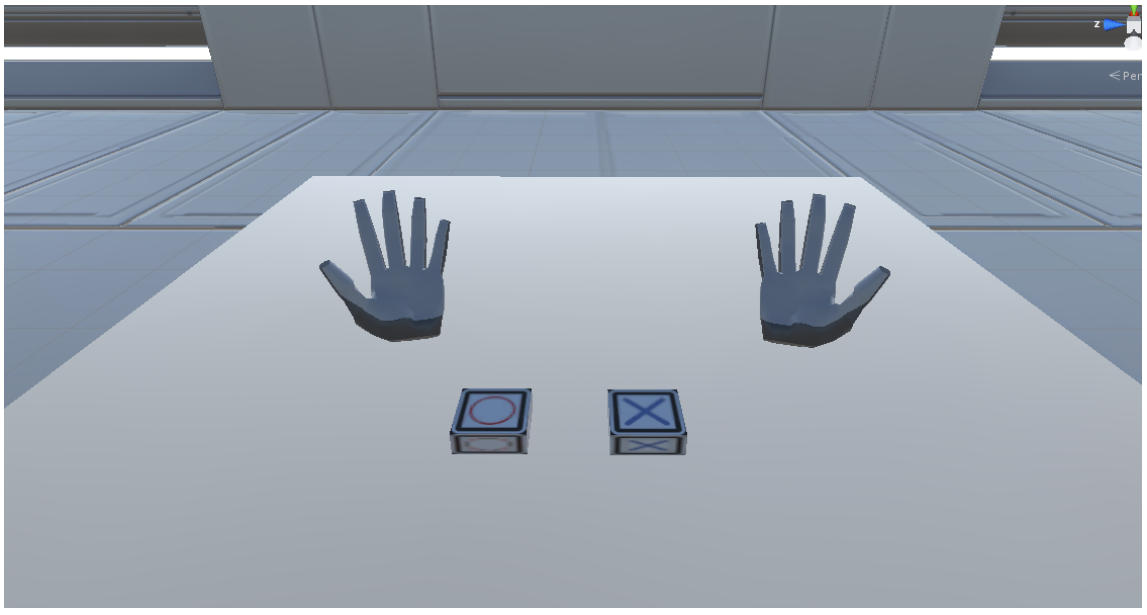

Figura 36: Piezas de *TicTacToe*

$$i = jugadaElegida / SquareMatrix.Width$$

$$j = jugadaElegida \mod SquareMatrix.Width$$

Figura 37: Operación realizada para obtener los índices

Una vez que conseguimos que la jugada de la IA sea mostrada en el entorno, es hora de averiguar un método para que el jugador pueda introducir su jugada desde el propio entorno, es decir, colocando la pieza en el tablero. Para ello, utilizando componentes asíncronos de Unity creamos una **función asíncrona** con ciclos infinitos (llamada **jugadaAdversario**) que cede su prioridad al resto de hilos del entorno mientras comprueba que no se cumpla una condición (*Task.Yield()*) y además mediante **await** esperaremos que esta función se complete, de forma que la ejecución del *TicTacToe* no continuará hasta que el ususario haya confirmado su movimiento. Tenemos que indicar que en la función **jugadaAdversario** durante los ciclos infinitos se comprueban 3 situaciones:

- **La pieza ha caído de la mesa**. Si la pieza ha caído de la mesa, entonces no podremos recogerla, por tanto, esta se destruye y se vuelve a crear delante del usuario.
- **Se ha pulsado el botón rojo**. Si se ha pulsado el botón rojo, entonces como explicamos en la sección 4.5.1 la pieza se destruirá (esté en el tablero o en cualquier otro sitio) y se creará delante del usuario.
- **La pieza se ha colocado en el tablero y el usuario ha pulsado el botón verde**. Si esto ocurre, terminará de ejecutarse los ciclos infinitos, por lo que finalizará la espera de la jugada del usuario.

A partir de la colocación de la pieza del usuario, obtenemos la casilla donde se ha colocado gracias a **PlaceObjectOnGrid** ya que una vez se termina de ejecutar, esta devuelve una referencia a la casilla. Por tanto, ya que cada *Square* posee su fila y columna (*i*,*j*), se hace la operación inversa a la fig.37, la cuál se muestran en la fig.38 para obtener un único número que corresponde con la casilla (jugada) elegida.

$$jugada = i * SquareMatrix.Width + j$$

Figura 38: Operación inversa realizada para obtener la jugada

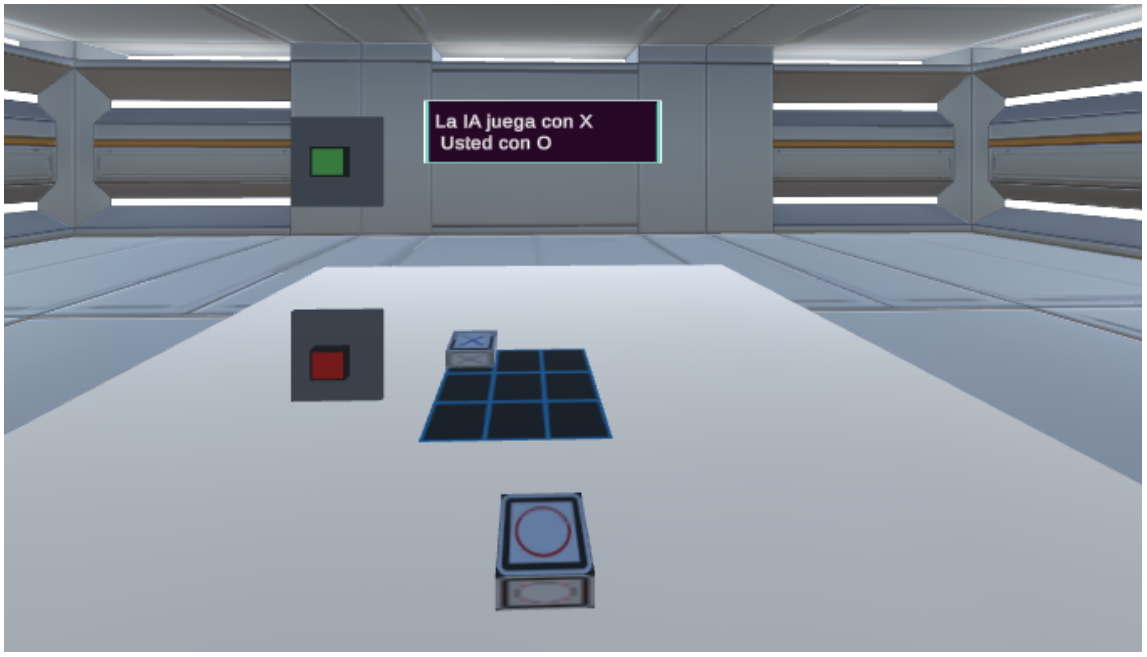

Figura 39: Simulación *TicTacToe*

Una vez teníamos hecho todo el sistema, faltaba una manera de comunicar al usuario instrucciones de la simulación, por lo que creamos el letrero visto en la fig.39 que nos dará ciertas indicaciones en cada simulación y además mostrará el resultado final de la partida. Este letrero será usado de aquí en adelante para todas las simulaciones y se activa y desactiva en el comienzo y fin de una simulación.

También indicar que se puede seleccionar el turno del usuario (*primero* o *segundo*) cuando decidimos iniciar la simulación en Alexa, como indicamos en la sección 4.4.3.

En conclusión, la simulación comienza desde Alexa e indicamos el orden, a continuación se crearán el tablero de 3x3, los botones y el letrero que tendrá una indicación. Cuando sea nuestro turno podremos pensar la jugada tantas veces como queramos y cuando tengamos una definitiva, pulsaremos el **botón verde**. Entonces la IA colocará la suya y cuando se llegue a un estado final (**EMPATE**, **GANA LA IA** o **GANA EL JUGADOR**) el letrero nos mostrará este resultado y habrá terminado la simulación.

4.5.3. Simulación Heurística Guiada

Para esta simulación creamos un tablero de 4x4 (e incluimos de nuevo los botones y el letrero) y le proporcionamos al jugador las piezas necesarias para completarlo, con lo que él deberá colocarlas evitando llegar a un estado que ya no tenga solución.

Además, el letrero poseerá una puntuación la cuál corresponde con la **heurística** del tablero en ese momento. Deberemos guiarnos por la puntuación para conocer cómo han sido nuestros movimientos o si hemos llegado a un estado final (bueno o malo), con lo que la simulación se detendrá. La especificación de esta heurística (*script* **FuncionHeuristica**) es la siguiente:

- double **Heurística**(*Square*[,] tablero)
- **Precondición**: Recibe la matriz de *Square*.
- **Postcondición**: Devuelve un *double* que podrá tomar valores entre 0 y el número de casillas del tablero (longitud total de la matriz); indicando 0 un **final erróneo** y el número de casillas un **final correcto**.

En esta y la siguiente simulación se ha utilizado la heurística indicada a continuación:

```
if(Estado_Inicial)
{
    heuristica = 1
}
else
{
    if(Casillas_Aisladas)
    {
        heuristica = 0
    }
    else
    {
        heuristica = Casillas_Ocupadas
    }
}
```

Esta heurística marcará como **final erróneo** (0) cuando alguna casilla se encuentre aislada, es decir, sólo cabe una pieza de 1x1, la cuál como no existe en nuestro entorno (el mínimo es 2x1) será imposible de rellenar. Si no, la heurística corresponderá en todo momento al número de casillas ya ocupadas, de forma que cuando se complete entero, tomará la longitud total de la matriz. Por tanto, la heurística seleccionada por nosotros guiará al jugador a rellenar el tablero.

Al comenzar la simulación, el jugador encontrará delante la primera pieza y el mecanismo para colocarla será el mismo al *TicTacToe*: para confirmar el movimiento deberemos pulsar el **botón verde**, tras esto, aparecerá la siguiente pieza y el jugador deberá elegir el movimiento. Si se ha llegado a un estado final al pulsar el botón, la siguiente pieza no aparecerá ya que ha terminado la simulación.

En este caso, las piezas que utilizaremos serán únicamente del **Tetris** y su orden de aparición será el siguiente:

1. **Pieza morada**
2. **Pieza naranja**
3. **Pieza roja**
4. **Pieza naranja**
5. **Pieza morada**

A la hora de esperar que el jugador coloque su pieza, hemos utilizado el mismo método de la función asíncrona en *TicTacToe*, es por ello que la pieza que tenemos que colocar sigue el mismo comportamiento que las fichas de este juego: si se cae de la mesa, reaparece; si pulsamos el botón rojo, se coloca delante del usuario y si se pulsa el verde se confirma el movimiento.

Por otro lado, el letrero mostrará una de las dos siguientes opciones cuando hayamos alcanzado un final:

- HEURISTICA 16: COMPLETADO
- HEURISTICA 0: GAME OVER

En conclusión, en esta **simulación guiada** obtendremos una serie de piezas en orden que deberemos colocar evitando que la heurística alcance 0, en este caso la heurística será 0 cuando haya al menos 1 casilla que esté aislada. Para confirmar movimiento pulsaremos el botón verde, para modificar uno pulsaremos el rojo y para guiarnos del progreso tendremos el letrero. En la fig.40 podemos ver la simulación cuando se ha llegado al **final correcto**.

4.5.4. Simulación Heurística Libre

Esta última simulación implementada es una extensión de la anterior: en este caso creamos un tablero de 6x6 y el proceso será el mismo, deberemos colocar las piezas siguiendo la puntuación devuelta por la heurística de ese momento. La gran diferencia en este caso es que las piezas no se mostrarán delante de nosotros, sino que la simulación es libre y deberá ser el usuario el encargado de seleccionar las piezas que crea convenientes (mediante Alexa) para llegar al estado correcto de la heurística. La **heurística** es la misma que la utilizada en la simulación anterior y corresponde a la función **Heurística** del *script* **FuncionHeuristicaLibre** con la misma especificación anteriormente indicada.

Figura 40: Simulación Guiada

Para esperar a que el usuario coloque la pieza hemos utilizado de nuevo la *función asíncrona* comentada en las simulaciones anteriores. Sin embargo, como la pieza no es proporcionada por la simulación, el usuario deberá hacerse cargo de ella, por lo que al pulsar el botón rojo únicamente colocaremos la pieza delante del usuario si esta se encuentra en el tablero (deshacer movimiento). Además, la *función asíncrona* tampoco comprobará si esta se ha caído de la mesa. En la fig.41 podemos ver la simulación iniciada.

4.6. Generación de objetos

En este punto, el proyecto posee una gran capacidad de interacción y diversas funciones. Sin embargo, quisimos añadir una característica que proporciona una gran usabilidad de cara a empresas que trabajen con este tipo de tecnologías.

La idea era que el usuario pida un objeto mediante Alexa, este se busque en una base de datos (de Internet) y por último se origine en el entorno con todas las características que poseen las piezas creadas por nosotros. En el ámbito empresarial esto permitiría una gran facilidad a la hora de modificar/introducir objetos que el usuario puede usar en las simulaciones. Por ejemplo, una persona del departamento de diseño podría mejorar la forma de una herramienta y subir la pieza mejorada a la base de datos; la próxima vez que un usuario requiera esta herramienta, ya poseerá las mejoras de diseño.

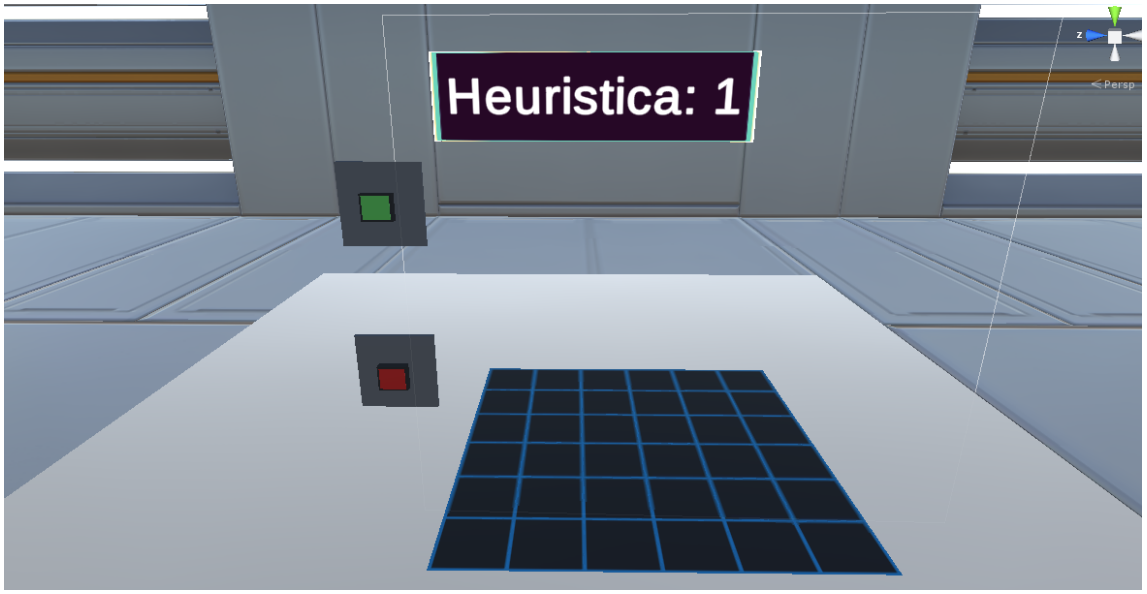

Figura 41: Simulación libre o no guiada

Para ello, la primera tarea era encontrar una manera de **incluir un objeto en Unity desde un archivo en tiempo de ejecución**. Tras esto, hallar una **colección de objetos 3D** que nos proporcione de manera sencilla los distintos archivos del objeto requerido y por último, **tratar estos objetos** en el entorno para que sean interactivos y posean características básicas como la gravedad. En esta sección trataremos en orden estos 3 aspectos.

4.6.1. UnityGLTF

Investigando sobre cómo introducir objetos en Unity desde archivo primero debíamos averiguar cuáles soportaba. Entre los tipos de archivos soportados se encuentran: **.fbx**, **.obj**, **.max** o **.blend**, siendo estos dos últimos de **Autodesk 3D Max** y **Blender** respectivamente.

Sin embargo, sin todavía poseer soporte oficial de Unity, encontramos el formato **.glTF**: un estándar abierto creado en 2015 y orientado a la distribución eficiente de escenas y modelos 3D ya que comprime el tamaño de ambos. Sobre este además existen dos repositorios en GitHub donde podemos encontrar *assets* que nos permiten en tiempo de ejecución crear objetos en nuestro entorno. Por tanto, debido a que posee soporte no oficial para Unity y que se orienta a la eficiencia en cargar modelos en tiempo de ejecución (buscamos siempre evitar sobrecargas en nuestro entorno) elegimos el formato **.glTF** para implementar la **generación de objetos**.

Como hemos comentado anteriormente, podemos encontrar dos repositorios similares que importan un objeto **.glTF** en tiempo de ejecución, los cuáles son **GLTFUti-**

lity[41] y **UnityGLTF**[42]. Nosotros nos decantamos por el segundo debido a estas razones:

- **Autores**. Los autores de **UnityGLTF** son los creadores del estándar **.glTF** (*Khronos Group*) por lo que estarán más capacitados y poseerán mayor conocimiento del formato a la hora de incluir mejoras o nuevas características al repositorio.
- **Documentación**. Este repositorio contiene información y ejemplos para poder implementar nuestras funciones; a diferencia de **GLTFUtility**, el cuál apenas posee indicaciones.
- **Capacidades**. Dado que ninguno de los dos tiene todavía una versión final (1.0), ambos poseen *bugs* y errores no solucionados; sin embargo, haciendo una prueba con estos dos en nuestro entorno pudimos comprobar que fue **UnityGLTF** el que respondía de manera más eficiente y provocando menos errores.

A partir de aquí, si importamos el *asset* que encontramos en su página, podemos tener acceso al componente encargado de introducir el objeto al entorno. Este componente denominado **GLTF Component** se colocará en un objeto vacío (que llamaremos *GLTFLoader*) de forma que los elementos que cree estarán indicados como sus hijos en la jerarquía y además recibirán el nombre que el usuario indicó a Alexa para crearlos.

En la fig.42 podemos observar los parámetros que podemos modificar de **GLTF Component** y cómo están configurados para que funcione en nuestro entorno. *Multithreaded* ha sido activado para evitar que, durante la carga de los distintos componentes del elemento, el entorno se detenga; *Use Stream* nos permite obtener una ruta de nuestro sistema en vez de una web y por último hemos marcado *Mesh Convex* en *Collider* para que las colisiones de los modelos 3D estén bien definidas y por tanto se cree una experiencia interactiva eficiente. Además se ha desactivado *Load On Start* para evitar que cada vez que iniciemos el entorno se genere un objeto.
Una vez habíamos instalado **UnityGLTF** en nuestro entorno, debíamos hacer varias pruebas (como podemos ver en la fig.43) para ver cómo se comportaba. Para ello, obtuvimos de Internet distintos modelos en formato **.glTF** y al crearlos se lograron buenos resultados. Sin embargo, encontramos 2 fallos (o *bugs*) principales:

- **Tiempos de carga**. Si el modelo posee demasiados polígonos o su tamaño supera un cierto límite, el tiempo de carga puede ser de varios minutos. En otras pruebas logramos crear varios objetos de una gran calidad y detalle en un tiempo razonable, por lo que puede que estos tiempos de carga se deban a otras circunstancias relacionadas con la implementación de **UnityGLTF**.

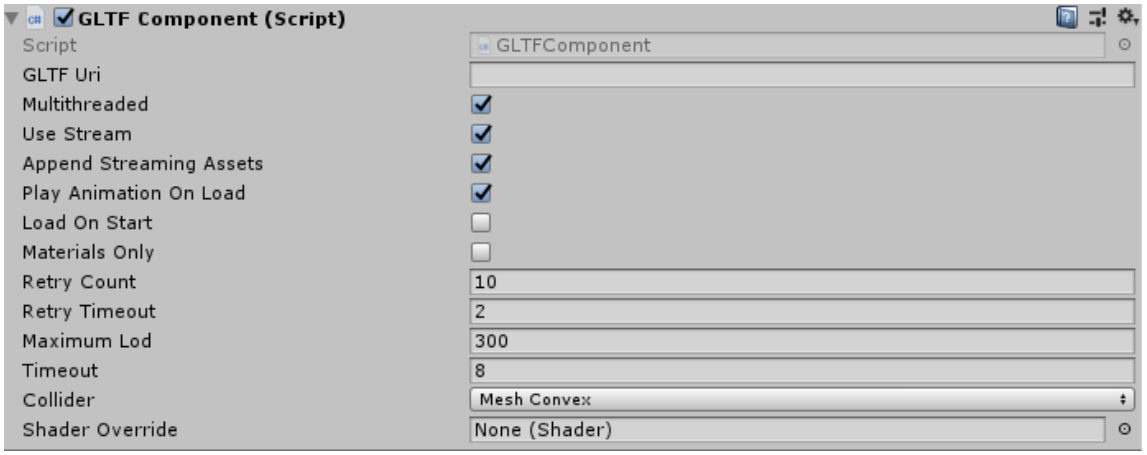

Figura 42: Parámetros del componente GLTF Component

- **Objetos no creados**. Aunque hay objetos que tardan bastante en crearse, otros ni siquiera llegan a terminar el proceso, ya que se devuelve el siguiente error: **ObjectDisposedException: Cannot access a disposed object. Object name: Stream has been closed**.

  Este error se ha reportado en diversas ocasiones en el apartado de *Issues* del repositorio y algunos usuarios apuntan que puede ser un fallo del código a la hora de importar el objeto. Sólo sucede con alguno modelos y siempre con los mismos y hay que añadir que desde el **visor 3D de Windows** estos modelos se pueden visualizar correctamente, pero a la hora de ejecutar **UnityGLTF** se retorna el error anteriormente mencionado, por lo que se descarta la posibilidad de que el archivo esté corrupto.

Por último, añadir que **UnityGLTF** nos permite también ejecutar animaciones que traigan los modelos que hayamos creado, lo que nos proporciona realismo en objetos como vehículos o animales.

4.6.2. Sketchfab

A la hora de buscar modelos 3D para la sección 4.2.2 recurrimos a diversas páginas como **Sketchfab**[43], **CGTrader**[44] o **Free3D**[45]. De todas estas, **Sketchfab** fue la que poseía más modelos, la más robusta y además posee API.

Sketchfab es una página web que permite visualizar, descargar y subir **modelos 3D** e incluso posee soporte para **Realidad Aumentada** y **Realidad Virtual**. **Sketchfab** posee una inmensa colección de objetos 3D gratuitos ya que los usuarios a la hora de subirlos pueden seleccionar esta opción. Pero lo más interesante y que nos hizo decantarnos por **Sketchfab** es que posee una API robusta y que cumple justo

Figura 43: Modelo 3D creado como prueba del funcionamiento de UnityGLTF

con nuestras necesidades: nos permite filtrar los modelos según varias características y a continuación la posibilidad de descargarlos en nuestro sistema.

Procedemos a explicar en detalle el funcionamiento de la API de **Sketchfab**. En la documentación[46] encontramos distintas secciones, pero ya que en el proyecto únicamente queremos descargar, accedimos a la sección *Downloading Models*. En esta se explica que para descargar un modelo mediante *HTTP* hay que seguir una serie de pasos en orden:

- **Autenticar el usuario**. Deberemos proporcionar un *identificador único* correspondiente a nuestra cuenta cada vez que hagamos la petición de descargar un modelo. Por ello, desde la sección *Settings/Password & API* podremos obtener nuestro *identificador único* o también llamado **API Token**.
- **Filtrar o buscar modelos**. La API, a través de HTTP, nos permite realizar varias operaciones como buscar modelos o subirlos. Es por ello que para este proyecto, ya que buscamos un modelo según la descripción del usuario, necesitaremos filtrarlos para obtener uno acorde. Para ello, la documentación indica que debemos realizar una operación GET a una URL que tiene una base tal que: **"https://api.sketchfab.com/v3/me/search?type=models"**. A partir de aquí, mediante parámetros que añadamos podremos filtrar los modelos que buscamos según las características de estos. En nuestro caso, utilizamos el parámetro **q** para solicitar un modelo con el nombre indicado por el usuario; **downloadable=true** para que el modelo sea gratuito y descargable y

pbr_type=false ya que disminuye la complejidad del modelo y genera mejores resultados en cuanto a la generación del mismo con **UnityGLTF**. Por tanto, si hacemos un GET a la dirección **"https://api.sketchfab.com/v3/search ?type=models&q=NOMBRE &downloadable=true &pbr_type=false"** se nos retornará un JSON con todos los modelos que han cumplido los parámetros indicados.

- **Solicitar una descarga**. Una vez que hemos seleccionado un modelo de entre los obtenidos en la búsqueda, debemos formar una URL que nos permita solicitar la descarga. Para ello debemos hacer una petición GET a una URL con la estructura: **https://api.sketchfab.com/v3/models/ {UID} /download** donde UID es un ID único que identifica a cada modelo y se puede encontrar en el JSON obtenido en el paso anterior. Además, deberemos añadir una cabecera "Authenticator" donde incluyamos nuestro **API Token** de forma que **Sketchfab** registre qué usuario ha realizado la petición. Si la petición ha sido exitosa, se nos devolverá un JSON con dos enlaces temporales para descargar el modelo en **.glTF** o en **.usdz**.

- **Descomprimir el archivo**. Una vez hemos descargado el archivo correspondiente al enlace temporal de **.glTF**, encontraremos un fichero *zip* el cuál al ser descomprimido tendrá los recursos necesarios para importar el modelo, es decir, a **UnityGLTF** deberemos pasarle la ruta de esta carpeta descomprimida.

Por último, debemos automatizar estos procesos, por lo que en nuestro entorno creamos el *script* **cargarArchivo** el cuál posee la función *insertarObjetoEscena(string name)* la cuál a partir de la cadena *name* buscará el primer objeto con ese nombre y mayor relevancia y lo introducirá al entorno. Para ello, la función está dividida en cada uno de los pasos comentados anteriormente: primero filtramos los modelos y obtenemos el **UID** del más relevante; a partir de este solicitamos la descarga y obtenemos el enlace temporal; con este último descargamos el archivo *zip* y descomprimimos los recursos en una carpeta auxiliar del proyecto ("*ZIP*", la cuál elimina su contenido cada vez que la función es invocada) y por último, nos queda la tarea de indicar a **UnityGLTF** la ruta del modelo para su creación y el posterior tratamiento del objeto. Durante todo este proceso activaremos el letrero y este indicará "*Cargando*" hasta que se haya creado el objeto, entonces pondrá "*Listo*" para desaparecer de nuevo en 3 segundos; sin embargo, si la búsqueda inicial no arroja ningún modelo 3D entonces se mostrará durante unos segundos el letrero con la frase "*No se ha encontrado el objeto especificado*" y a continuación terminará la ejecución de la función.

Cabe mencionar un par de aspectos, el primero de ellos es que la carpeta auxiliar "*ZIP*" se creará en el directorio origen del proyecto al inicio de la ejecución del mismo, para asegurar que se encuentre disponible a la hora de descargar los archivos *zip*.

El segundo es que debido a que hemos seleccionado el **plan gratuito de Sketchfab**, únicamente podremos realizar 50 peticiones cada 24 horas y si se ha excedido el número de estas, capturamos la excepción y mostraremos al usuario: "*Límite de peticiones alcanzado*".

4.6.3. Tratamiento del objeto

Una vez hemos averiguado como obtener desde una colección de objetos 3D un modelo que pueda ser descargado en nuestro proyecto, nos queda la tarea de generar ese objeto y que funcione correctamente de cara al usuario.

Primero, simplemente debíamos indicarle al componente *GLTF Component* la ruta del modelo e invocar la función correspondiente que iniciara el proceso de creación. Sin embargo, en pruebas iniciales descubrimos un problema: **cada modelo 3D viene con su propia escala**. Esto quiere decir que si al modelo le asignamos *0.002* de escala en los 3 ejes, puede ocurrir que algunos objetos 3D queden a un tamaño razonable, mientras que otros sean extremadamente grandes (o pequeños). Para solucionar esto, encontramos un método ideal: el usuario será el encargado de elegir la escala (tamaño) que mejor le convenga según sus preferencias, de forma que antes de que el objeto sea definitivo, mediante algún tipo de input este seleccionará el tamaño adecuado.

Para este input hemos utilizado una **barra deslizadora** de **Leap Motion** de forma que mediante nuestras propias manos, arrastrando un cursor, podemos colocar un valor entre dos límites. Además, para confirmar el valor que posea el objeto en ese momento, usaremos el botón verde de las simulaciones de este proyecto. Como podemos ver en la fig.44, la barra posee un cursor amarillo que coloca la escala entre 0 y 0.5 (en los 3 ejes) de forma que deberemos pulsar y arrastrar ese cursos para elegir un tamaño. Hemos colocado de nuevo un método asíncrono de ciclos infinitos hasta que el usuario pulse el botón verde y en cada momento actualizamos el valor del objeto al de la *barra deslizadora* para que el usuario pueda comprobar cómo quedará con la escala seleccionada. Cabe mencionar también que el botón, la barra deslizadora y el letrero desaparecerán tras haber completado la creación del objeto. Una vez se ha seleccionado la escala y se ha creado el objeto, debemos añadirle los componentes necesarios, en nuestro caso son los siguientes:

- **Rigidbody**. El cuál nos permite añadir la gravedad y hacer que se produzcan las colisiones correspondientes con otros objetos.
- **InteractionBehaviour**. Es el componente que gestiona las acciones con **Leap Motion**. Gracias a este componente podremos interactuar con nuestras manos.

Tras esto, habrá terminado la ejecución de la función *insertarObjetoEscena* y habremos incluido un objeto nuevo al entorno.

Figura 44: Selección de escala en un modelo 3D

Cabe mencionar que si el modelo 3D arroja durante el proceso de creación la excepción anteriormente mencionada (**ObjectDisposedException**), el programa la capturará, mostrará en el letrero "Error al cargar el archivo" y se terminará la ejecución de la función, para que el usuario a continuación pueda buscar otro objeto.

4.6.4. Invocación mediante Alexa

Esta característica de generar un objeto debe ser invocada mediante Alexa, como vimos en la sección 4.4.3, con el *Intent Crear Objetos: "Alexa busca/genera un objeto {Objeto}"*; siendo *Objeto* un slot creado por nosotros.

Durante el desarrollo hemos modelado este *intent* para que cualquier palabra pueda ser introducida (no se ha colocado una validación), independientemente de que haya sido incluida o no en el slot *Objetos*, esto quiere decir que se puede pedir prácticamente cualquier objeto; dependerá de la disponibilidad de *Skecthfab* y del funcionamiento de *UnityGLTF* que el modelo de este sea añadido finalmente al entorno. Sin embargo, hemos seleccionado una serie de palabras que arrojan unos buenos resultados y las hemos añadido a *Objetos* para que el usuario pueda comprobar el potencial de esta característica. Las palabras son las siguientes:

- **Coche**.
- **Guitarra**.
- **Perro**.
- **Nintendo**.
- **Barco**.
- **Tractor**.
- **Herramientas**.
- **Avión**.
- **Comercio**.
- **Oficina**.

Muchas de las palabras están relacionadas con el mundo empresarial o industrial, sin embargo, otras como *Nintendo* o *Guitarra* están incluidas para demostrar el potencial de esta característica y que el usuario pueda ver que las opciones son prácticamente infinitas.

Para asegurar una mejor búsqueda de modelos tanto en calidad como cantidad, dado que **Sketchfab** es una página en inglés, la mayoría de sus modelos están categorizados en este idioma. Es por ello que a la hora de buscar las palabras de este slot, desde el código de la *skill* de Alexa se traducen al inglés, de forma que antes de realizar la búsqueda de "perro" se introducirá la palabra "dog". Sin embargo, las palabras que sean buscadas y estén **fuera del slot** (como por ejemplo "ajedrez") serán introducidas en la petición tal y como se recibieron.

4.7. Optimización de la aplicación

Nuestra aplicación posee un gran potencial en su versión definitiva y esto hace que tenga una serie de características que consumen gran cantidad de recursos de nuestro sistema, como puede ser el componente que controla el casco VR *Oculus Rift*, los *assets* de *Leap Motion* o el proceso que conlleva generar un objeto desde cero. Además, la aplicación posee varios procesos que se ejecutan en segundo plano como el *script* encargado de recibir el mensaje de la cola en la nube o la función que espera la jugada del usuario en el *TicTacToe*. Dado que queremos minimizar el consumo de recursos que utiliza la aplicación para así poder garantizar su uso en un gran número de equipos, decidimos recurrir a **métodos de optimización**. Para ello, nos apoyamos en un tutorial oficial de Unity llamado "*Optimizing you VR/AR Experiences*"[47] que contiene varios métodos que nos permiten optimizar nuestro entorno. Pasamos a explicar los que hemos utilizado.

El primer método utilizado consiste en crear un único **perfil de calidad de gráficos** que contenga lo esencial para nuestra aplicación. En este hemos decidido desistir del **Anti-Aliasing**, reducir el número de píxeles de luz, desactivar los reflejos renderizados en tiempo real e incluso minimizar la calidad y cantidad de las sombras. Esta última característica fue seleccionada debido a que en nuestro entorno apenas se utilizan luces y sombras ya que hemos optado por centrarnos en el apartado técnico

a costa de disminuir en pequeña medida el apartado estético. A este perfil le hemos denominado "*Fastest*" y todas las características seleccionadas pueden observarse en la fig.45.

A continuación decidimos seleccionar como **método de renderizado** para VR el **single-pass** ya que procura utilizar los objetos que ya están renderizados en ambos ojos, a diferencia del método **multi-pass** el cuál intentará renderizar los elementos por duplicado. El método **single-pass** consigue una gran actuación a cambio de una peor calidad de luces, sin embargo, debido a que las luces y las sombras de nuestro entorno no son algo primordial, optamos por este. Además, para conseguir una mayor accesibilidad en diversos equipos, decidimos no seleccionar el método **single-pass instanced** ya que está únicamente disponible en Windows y hace que el resto de sistemas operativos ejecuten **multi-pass** en su lugar.

El siguiente método utilizado se denomina "**Light Baking**" y consiste en colocar la etiqueta *Static* a los elementos que no modifiquen su posición durante la ejecución de la aplicación y así precalcular la incidencia de las luces. Este método ahorra cálculos en tiempo de ejecución, por lo que podremos evitar la saturación de la aplicación en los momentos que requieran más recursos. Como hemos comentado anteriormente, para realizar **Light Baking** hay que colocar la etiqueta **Static**, desde el editor del componente, a todos los objetos inmóviles. A continuación, debemos colocar la luz de nuestro entorno como "*Baked*" y en la pestaña **Window - Rendering - Lighting settings** seleccionamos "*Generate Lighting*" y esperamos a que se complete el proceso. Tras esto, habremos generado las luces de todos los objetos marcados como "*Static*" para que después sean usadas durante la ejecución de la aplicación.

Por último, utilizamos el método conocido como **Static Batching** el cuál es similar al anterior, ya que consiste en activar esta característica en nuestro proyecto y a continuación marcar todos los objetos inmóviles con la etiqueta **Batching Static**. Este método hace los elementos marcados con esta etiqueta sean unidos en un gran *mesh*, es decir, agrupa los datos de *forma*, *tamaño* o *posición* para obtener un único objeto que los representa, de esta forma se ahorra en memoria ya que los fondos están compuestos de una gran cantidad de sub-objetos que al activar esta característica se formarán en uno solo durante la ejecución.

4.8. Build

Una vez está todo terminado, queda crear los archivos correspondientes que compondrán la aplicación. Para ello, colocamos como "*Company Name*": **RafaRoman**, la versión es **1.0** y hemos decidido seleccionar como "*Product Name*": **LeapVR**. Tenemos que mencionar que hemos añadido añadido desde las opciones de Graphics en Unity los *shaders* **PbrMetallicRoughness** y **PbrSpecularGlosiness**, los cuáles

son propios de **UnityGLTF** y sin ellos no podremos utilizar la característica de *Generación de Objetos* tras la *build*. Estos *shaders* han aumentado considerablemente el peso de los archivos finales.

Tras todo esto, una vez construida nuestra aplicación, se genera una carpeta que ocupa alrededor de **1.2 Gb**. Al ejecutar **LeapVR.exe** nos encontramos con una ventana predeterminada de Unity (como vemos en la fig.47) que nos permite seleccionar **la calidad de la aplicación**, **la resolución**, **el monitor** e incluso modificar **el input** que utilizaremos (en este caso no se utiliza un input físico). Como comentamos anteriormente, el único perfil de calidad para ejecutar nuestra aplicación es "*Fastest*", creado especialmente para garantizar una ejecución eficiente, y la resolución dependerá de las configuraciones que cada usuario posea y haya colocado en su sistema; en nuestro caso ejecutaremos la aplicación en **1920x1080** y Pantalla Completa.

Cuando hayamos seleccionado todo podremos presionar el botón "Play!" y a continuación se nos mostrará una pantalla con "**Made in Unity**" como marca de agua, ya que como explicamos en la sección 3.2.2, poseemos la licencia **Unity Personal**. Tras esta, accederemos finalmente a nuestra aplicación.

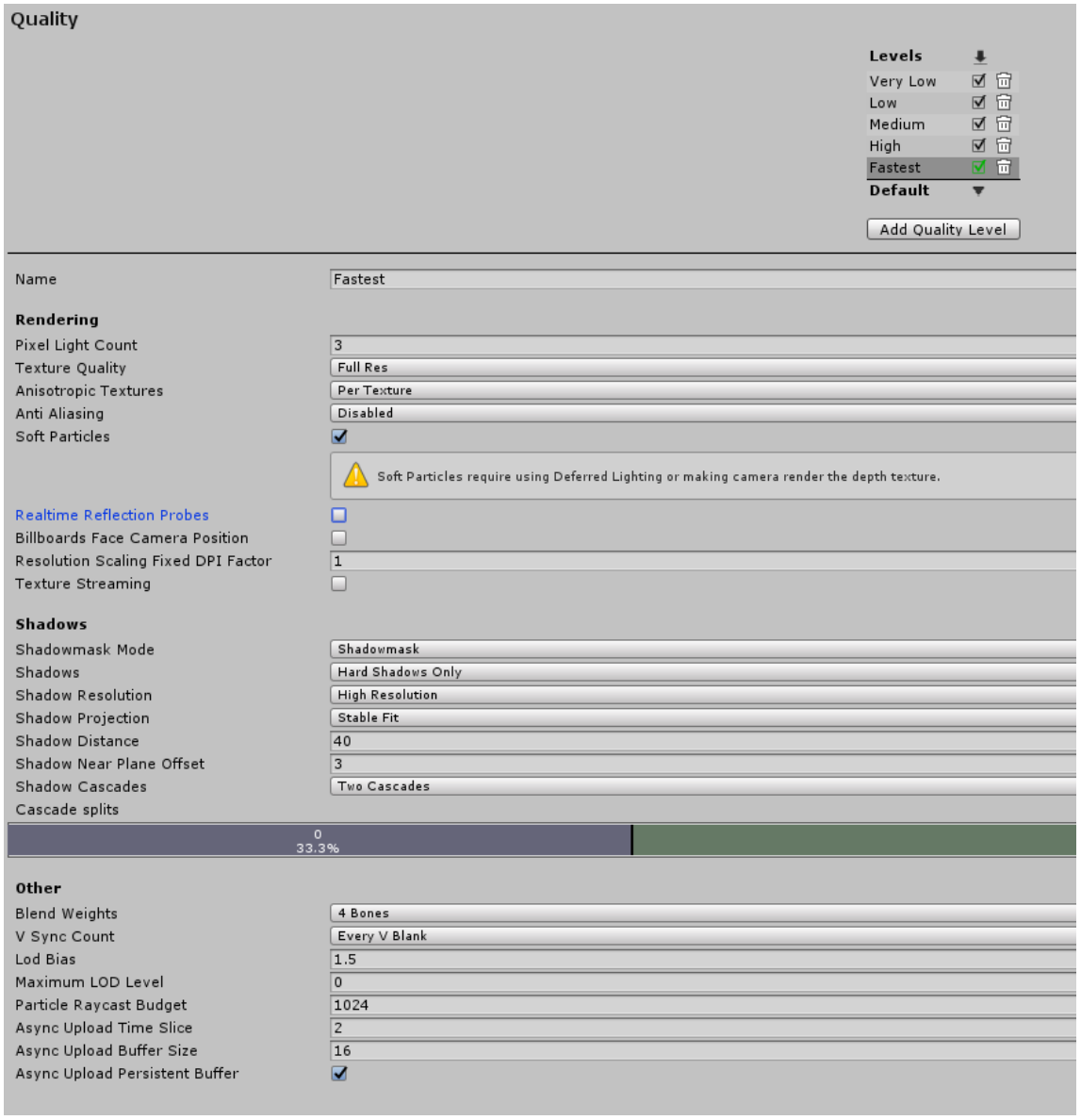

Figura 45: Perfil de calidad de gráficos utilizado en la aplicación

5. Conclusiones

Este proyecto recoge de manera concisa las bases para crear una aplicación virtual (con las características descritas) y el conocimiento suficiente para que el lector pueda elegir cómo modelarla y qué modificaciones añadirá. En el marco teórico que recoge se especifica el funcionamiento de cada una de las tecnologías de manera que el usuario podrá conocer en un primer momento para qué sirve cada una y cómo podrá usarlas. Por lo que este proyecto no sólo funciona como guía o marco de referencia para un usuario experimentado, sino también para personas que comiencen a investigar desde cero sobre la creación de aplicaciones virtuales. Por estas razones la etapa de desarrollo del proyecto fue redactada de manera **cronológica**, ya que produce una dualidad oportuna. Por un lado, si un usuario experimentado desea leer este proyecto, visionará el desarrollo de la manera en la que se produjo, comprendiendo y entendiendo cómo se realizaron los avances y qué necesidades fuimos encontrando en el proceso. Por otro lado, si el lector es novel en los temas tratados, podrá usar este proyecto como manual de consulta para así crear paso a paso una aplicación similar.

Este trabajo ha supuesto una gran adquisición de habilidades a la hora de resolver problemas ya que el desarrollo del mismo se produjo sin conocer las tecnologías utilizadas o si sería posible crear algo de las capacidades que hemos logrado, debido a que se comenzó como un proyecto del programa de *alumno colaborador*. Estas habilidades nos permitirán en un futuro trabajar con una visión más amplia en cualquier característica que debamos implementar. Además ha supuesto un gran proceso de investigación de las distintas tecnologías y sobre todo de la **Realidad Virtual**, gracias a la creación de este proyecto somos capaces de utilizar Unity para realizar aplicaciones de una cierta complejidad y que estas puedan ser visualizadas a través de un casco VR. Remarcar también que hemos obtenido conocimientos acerca de tecnologías emergentes y de gran capacidad como pueden ser **Leap Motion**, **Alexa** o **AWS Lambda**.

Comentando las aportaciones que produce este trabajo al sector debemos mencionar que en todo momento la idea del proyecto es crear un prototipo que una empresa utilice en labores de aprendizaje, enseñanza o prácticas; de forma que el usuario que use la aplicación virtual interprete los conocimientos adquiridos en esta como un entrenamiento previo al proceso real. La razón de elegir estas tecnologías es que crean una experiencia muy verosímil al usuario, por lo que las simulaciones permitirán a la empresa demostrar el funcionamiento de un sistema sin poner en riesgo al trabajador y ahorrando costes de materiales. Las simulaciones creadas en este proyecto sirven como base o punto de partida para que las empresas modelen las suyas propias según el proceso que elijan incluir, es decir, la función que cumplen nuestras simulaciones (y las características incluidas) es demostrar el potencial que alcanza nuestra aplicación virtual, y a raíz de esto, cada institución elegirá los procesos que desea simular.

Por tanto, este proyecto pretende impulsar la inclusión de la **Realidad Virtual** y sistemas similares en el **ámbito laboral o empresarial**, de forma que se eviten gastos y se gane en integridad para los trabajadores; todo esto consiguiendo una infraestructura de precio ínfimo.

Valorando el resultado final de este proyecto hemos de decir que se ha conseguido una aplicación virtual que ha superado con creces las expectativas iniciales. En un primer momento la idea era crear una aplicación virtual que fuera interactiva con Alexa y que dentro de esta aplicación se pudiera incluir *Oculus Rift* y *Leap Motion*. Tras conseguir esto, las funciones que trajo consigo el **tablero** y las posteriores **simulaciones** consiguieron una experiencia interactiva completa. Por último, la **generación de objetos** añade más potencia al entorno, ya que por un lado los usuarios podrán requerir objetos de manera directa y fácil y por el lado de las empresas estas podrán subir los modelos de manera sencilla a su base de datos. Por tanto, pensamos que este trabajo supone un proyecto interesante y poco común que podrá servir como una aplicación final con uso en empresas o incluso como base o punto de partida a aquellos que quieran crear un proyecto similar.

Otro aspecto importante a tener en cuenta es el **presupuesto** del proyecto, ya que además de poseer grandes capacidades como hemos comentado anteriormente, este ha sido realizado buscando el mínimo coste para el desarrollo. Por tanto, el presupuesto ha sido el siguiente:

- **Oculus Rift**, 600€
- **Leap Motion**, 100 €
- **Sistema capaz de ejecutar la aplicación**, 600€ (el sistema con menor capacidad que logró ejecutarla)
- **Adaptador Leap Motion VR**, 0 € (impresora 3D)
- **Amazon Echo Dot** (Alexa), 60€
- **TOTAL:** Aproximadamente 1360€

Cabe mencionar que no hemos añadido en el presupuesto el coste por mantener en ejecución los sistemas en la nube (**CloudAMQP** y **AWS Lambda**) ni la licencia de Unity (utilizamos **Unity Personal**) ya que para el desarrollo seleccionamos el plan gratuito de todos estos. Sin embargo, si una empresa pretende usar nuestro proyecto de manera intensiva, este ejecutará muchas peticiones a los sistemas y por tanto deberá adquirir alguno de los planes ofertados por ambas compañías.

El presupuesto mostrado anteriormente fue el utilizado para realizar el trabajo con los elementos que teníamos (adquiridos entre 2018 y 2019). Sin embargo, como comentamos en la sección 2.2, en Octubre de 2020 durante la realización de este proyecto, *Oculus* lanzó **Oculus Quest 2**, un casco VR *standalone* que supone el dispositivo

ideal para desarrollar este trabajo, ya que costando únicamente 350€ nos ofrece las siguientes mejoras:

- **Menor número de cables.** Ahora el casco VR Quest 2 no requiere ninguna conexión a nuestro ordenador (aunque opcionalmente podemos) ni la colocación de sensores, por lo que montar un equipo será más sencillo y requerirá menos espacio.
- **No necesita de *Leap Motion***. Al poseer una serie de cámaras que nos permiten el rastreo de nuestras manos, *Leap Motion* queda en una posición obsoleta, ya que el propio hardware del casco realiza la función de este dispositivo, poseyendo soporte para Unity.
- **Precio total del equipo**. Este proyecto requirió el casco VR *Oculus Rift CV1*, *Leap Motion*, el *Amazon Echo Dot* y un sistema capaz de ejecutar nuestra aplicación, lo que ronda los 1360€ como hemos comentado anteriormente. Sin embargo, utilizando **Oculus Quest 2** únicamente necesitamos el casco y el **Echo Dot**, ya que un ordenador externo es simplemente opcional y el rastreo de manos viene incorporado en sus cámaras. Este casco en su versión de 64GB cuesta 350€, por lo que si hubiéramos desarrollado este proyecto en el mismo el total sería de unos 410€, a diferencia de los 1360€ de nuestro equipo; lo que significa que habríamos sustituido un equipo completo por un único casco VR más el **Echo Dot**, y además habríamos abaratado el precio de manera exponencial.

Por último, como **trabajo futuro y mejoras** vamos a comentar tres aspectos. El primero de ellos es la incorporación al entorno de otro sistema, similar al tablero, de forma que se permitan interacciones diferentes y a partir de ello, simulaciones nuevas, ya que debido a la naturaleza del tablero estas conservan un elemento común o general entre ellas. Es decir, añadir otros sistemas nos permiten crear simulaciones capaces de mostrar nuevas experiencias al usuario. El segundo aspecto a tratar como **trabajo futuro** es aumentar la capacidad de Alexa de informarnos ante distintas situaciones. Esto quiere decir que si nuestro entorno ha devuelto un error, este sea enviado a Alexa y se nos comunique al momento; aunque no siempre debe ser un error, la idea es que Alexa nos informe de ciertos aspectos tales como la heurística de la simulación actualmente o si el objeto que queremos generar no ha arrojado ninguna búsqueda. El tercer y último aspecto es la incorporación de un método que nos permita usar la *skill* de manera individual, para así poder activarla públicamente y por tanto usarla junto con nuestra aplicación, ya que actualmente si la *skill* estuviera pública y varios usuarios descargaran nuestro proyecto, las acciones que cada uno le indique a Alexa se ejecutarán en cada uno de los entornos y por tanto se mezclarán. La solución ante esto podría ser la generación de un código de autenticación mostrado desde nuestro proyecto y que cada usuario introduzca por voz o desde la app a la *skill* antes de comenzar a usarla. Esto supondría escalar la aplicación para poder

permitir varios usuarios, por lo tanto habría que modificar la infraestructura y los servicios en la nube.

6. Anexo: Manual de Usuario

A. Introduccion

Este manual pretende recoger de forma concisa los requisitos y especificaciones de nuestra aplicación de forma que podamos guiar a un usuario a través del proceso de instalación. Además, una vez se hayan reunido los conceptos necesarios, pasaremos a explicar el funcionamiento de la misma y las distintas opciones que tendremos disponibles a realizar.

B. Requisitos

Antes de tratar el tema de software debemos indicar que obviamente necesitamos el siguiente equipo para ejecutar nuestra aplicación:

- **Ordenador**
- Casco VR **Oculus Rift CV1** o similares de **Oculus**
- **Leap Motion**
- **Dispositivo Alexa**
- **Adaptador Leap Motion VR**, ya que nos permite fijar el dispositivo Leap Motion a nuestro casco VR

Para el caso del ordenador, y guiándonos por los requisitos mínimos dictados por **Oculus** para el correcto funcionamiento de sus cascos, recomendamos al menos las siguientes características:

- **Capacidad RAM** de **8Gb**
- Disco duro sólido (**SSD**) de **128 Gb**
- 3 puertos **USB 3.0**
- Gráfica **Nvidia GTX 970**
- Sistema Operativo **Windows**.

Para el caso del casco VR **Oculus** deberemos tener instalada la aplicación *Oculus* (desde el enlace `https://www.oculus.com/setup/?locale=es_ES`) ya que esta controla el funcionamiento del mismo y provee los drivers necesarios para correr aplicaciones de Realidad Virtual; y para **Leap Motion**, el **Leap Motion Control Panel** (enlace: `https://developer.leapmotion.com/sdk-leap-motion-controller/`) que actúa de igual manera que la aplicación de *Oculus*.

Por último, para el caso de Alexa requerimos cualquier dispositivo que la soporte (Echo Dot, Echo Show, étc.) y tener instalado en este la *skill* **Objetos V. R.** Para ello, simplemente tendremos que acceder a la aplicación móvil **Amazon Alexa** en la que hayamos vinculado este dispositivo y en la sección *Skills y juegos* deberemos introducir en el buscador **Objetos V. R.** y seleccionar *Permitir su Uso*. (La skill de nuestra aplicación no se encuentra activa actualmente debido a razones del desarrollo del proyecto).

C. Instalación

Una vez hemos reunido el material indicado e instalado las aplicaciones necesarias, podemos pasar a comentar los archivos que componen nuestro proyecto. Para utilizar nuestra aplicación debemos descargar una carpeta que ocupa **1,21 Gb**, por lo que recomendamos al menos tener **5 Gb** disponibles en nuestro disco duro antes de descargarla para así asegurar un correcto funcionamiento.

La jerarquía de la carpeta origen (**LeapVR**) se muestra en la fig.46, aunque en esta el único archivo que nos interesa y que comienza la aplicación es **LeapVR.exe**. Debido a que la característica de **Generación de Objetos** debe descargar los archivos necesarios, nuestro proyecto crea la carpeta auxiliar "*ZIP*" dentro del directorio origen, donde los almacena temporalmente para después eliminarlos; por ello, **LeapVR.exe** necesita ser **ejecutado como administrador** para asegurar la correcta creación/eliminación de la carpeta "*ZIP*" y así obtener la ejecución deseada. Además, cabe mencionar que como se indica en varios foros oficiales, algunos antivirus como **Avast** no reconocen los programas creados en Unity como **seguros** y por tanto los eliminan al momento de abrir el archivo ejecutable, por lo que recomendamos desactivar previamente nuestro antivirus y volver a activarlo una vez hayamos cesado la ejecución de esta aplicación.

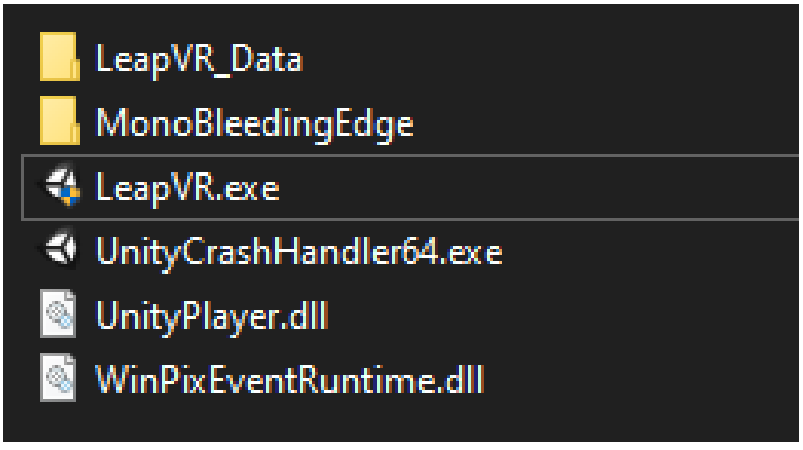

Figura 46: Carpeta raíz de LeapVR

D. Uso

Al momento de ejecutar **LeapVR.exe** deberemos esperar unos segundos a que la aplicación se inicialice y a continuación se nos mostrará la ventana de la fig.47, donde podremos elegir la resolución, la pantalla donde deseamos que se muestre y si queremos que la aplicación esté en modo ventana; por otro lado, la única calidad de los gráficos disponibles es **Fastest** (ya que mejora la ejecución de la aplicación) y el **Input** no es necesario modificarlo ya que el único input disponible será a través de **Leap Motion**.

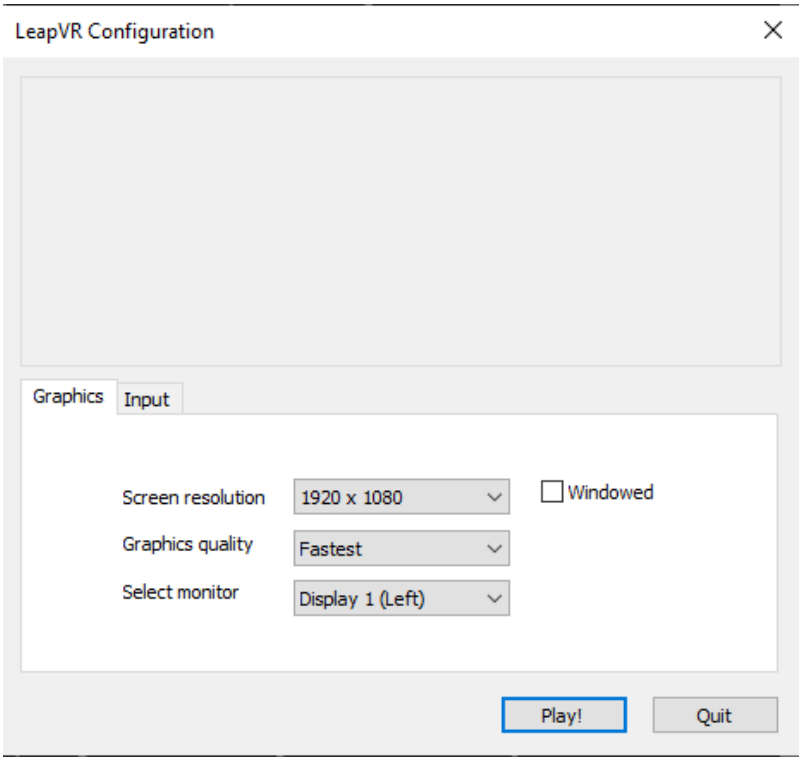

Figura 47: Ventana inicial mostrada en la aplicación

Una vez hayamos elegido la configuración, pulsamos **Play!** y se nos mostrará una marca de agua "Made in Unity!", tras esta habremos accedido a nuestra aplicación y nos encontraremos en un fondo de temática *Sci-fi* similar a una nave espacial y frente a nosotros una mesa. Si levantamos las manos y las ponemos delante del visor podremos observarlas a través de los modelos de **Leap Motion**.

En nuestro proyecto la mayoría de objetos están disponibles para interacción mediante **Leap Motion**, por tanto, a través de nuestras manos podemos realizar acciones como agarrar (cerrando el puño dentro de la pieza), levantar o incluso empujar. Sin embargo, debemos mencionar que las manos existen en el entorno mientras estas

están siendo visualizadas por **Leap Motion**, por tanto, en el momento en el que el usuario deja de mirar hacia sus manos, estas desaparecerán, eliminando cualquier interacción. Por ejemplo, si queremos lanzar un objeto, deberemos agarrarlo y a la hora de mover nuestro brazo hacia detrás, debemos seguirlo con la mirada (ya que en la parte delantera del casco se encuentra *Leap Motion*) y entonces será hora de mover el brazo hacia delante y en la trayectoria abrir la mano para soltar el objeto (todo esto siguiendo con la mirada también).

Desde este momento, queda a elección del usuario las acciones que tomará a continuación y estas deberán ser activadas a través de **Alexa**. Las acciones disponibles son las siguientes:

- **Crear una pieza**: mediante una frase como "*Alexa, abre Objetos V. R. y crea una pieza de color ——* ", siendo los colores disponibles: **Azul**, **Amarillo**, **Rojo**, **Verde**, **Morado** y **Naranja**, cada uno de estos supone un tipo de pieza distinta. Las piezas creadas mediante este método contendrán un cartel indicativo con el ID de la misma y cada una de ellas poseerá uno distinto.

- **Eliminar una pieza**. Se activa con la frase: "*Alexa, abre Objetos V. R. y elimina la pieza con ID –* " y a continuación indicaremos el número marcado en la pieza que deseamos eliminar. Esto hará que la pieza desaparezca sin importar su posición.

- **Cambiar el entorno.** Consiste en cambiar el fondo en el que nos encontramos, tenemos 3 disponibles y la forma de pasar de uno a otro es cíclica. El primero es el inicial, el segundo consiste en una nave industrial con elementos acordes a la misma y el tercero es el exterior de una parcela también de carácter industrial, pero esta vez encontramos en la mesa próxima a nosotros diferentes herramientas como un martillo o unos alicates los cuáles están disponibles para interactuar con ellos. Para realizar esta acción indicaremos lo siguiente: "*Alexa, abre Objetos V. R. y cambia el entorno*".

- **Buscar y generar un objeto**. Esta acción permite buscar el modelo 3D (más relevante) de un objeto en la web *Sketchfab*, descargarlo e incluirlo al momento en el entorno. Además, como podemos ver en la fig.48, a través de una barra deslizadora (botón amarillo) podremos seleccionar la escala (tamaño) del objeto y a continuación, pulsando el botón verde estará disponible como objeto interactivo en el entorno. Para invocar esta acción deberemos indicarle a Alexa: "*Alexa, abre Objetos V. R. y genera un objeto ———* ", los **objetos que recomendamos** son: **Coche, Guitarra, Perro, Nintendo, Barco, Tractor, Herramientas, Avión, Comercio y Oficina**, pero podemos indicar cualquiera que deseemos. Tenemos que mencionar que esta petición sólo podrá realizarse **50 veces cada 24 horas** debido a políticas relacionadas con *Skecthfab*.

Figura 48: Generación de Objetos

Además de todas las acciones mencionadas anteriormente, podemos iniciar 3 simulaciones que tienen como base un **tablero**, **2 botones** y un **cartel**, como podemos ver en la fig.49. Este tablero está diseñado para que al colocar una pieza esta se magnetice en una posición, de forma que el usuario no tenga que preocuparse por colocar las piezas con gran precisión; simplemente poniendo la pieza de la forma que queremos, el tablero actuará y las colocará ajustándose a la malla. Para colocar una pieza correctamente evitando un funcionamiento erróneo debemos dejarla caer a una cierta altura en la posición seleccionada, evitando agarrar la pieza mientras esta toca el tablero. El funcionamiento de los botones es el siguiente: el **verde** sirve para confirmar un movimiento y pasar a la siguiente fase de la simulación, es decir, solo funciona si hemos colocado una pieza en el tablero, y el **rojo** para deshacer uno, es decir, si hemos colocado la pieza en una posición no deseada, pulsaremos el botón rojo y esta aparecerá delante nuestra; también podemos utilizar el botón rojo para traer de nuevo a nosotros la pieza si en algún momento ha llegado a una posición lejana a nosotros. Además, poseemos un cartel que nos informa de distintas cosas durante las simulaciones.

A continuación vamos a explicar las 3 simulaciones disponibles:

- **TicTacToe**. Esta primera simulación consiste en el famoso juego del **Tres en Raya**, pero en este caso jugamos contra una Inteligencia Artificial. El usuario controla la pieza con un **círculo** y la IA la pieza con una **cruz**; la pieza del usuario aparece delante del mismo en cada movimiento y tras este la IA coloca la suya. El juego finaliza cuando ha habido empate o alguno de los dos ha

Figura 49: Tablero, botones y cartel usados en las simulaciones

ganado. Para invocar esta simulación diremos: "*Alexa, comienza simulación de tictactoe/tres en raya ———* " y podremos elegir nuestro turno indicando **primero** o **segundo**. Podemos observar esta simulación en la fig.50.

- **Simulación Guiada**. Esta simulación consiste en rellenar el tablero de 4x4 casillas con las piezas dadas y guiarnos por una puntuación (heurística) que nos indica como de cerca nos encontramos de la solución. En este caso, el objetivo es rellenar el tablero completamente, por lo que si llegamos a un estado donde al menos un cuadrado ha quedado vacío, habremos fallado la simulación y deberemos reiniciarla. Esta se denomina "*guiada*" porque las piezas se proveerán al usuario, por lo que cada vez que se coloque una (y pulsemos el botón verde) se actualizará la puntuación y aparecerá la siguiente pieza a colocar. Si llegamos a un estado donde la heurística es 0 (fallamos la simulación) dejarán de aparecer piezas y se nos mostrará "*Heurística 0: GAME OVER*". En caso de llegar a la puntación máxima se mostrará "*Heurística 16: COMPLETADO*". Para activar esta simulación debemos decir a Alexa lo siguiente: "*Alexa, abre Objetos V. R. y comienza la simulación guiada*".

- **Simulación Libre**. Esta última simulación es una extensión de la anterior, ya que deberemos rellenar un tablero de 6x6 a través de una puntuación, pero en este caso utilizando las piezas que nosotros deseemos. Es decir, tendremos los mismos elementos que en la "*guiada*" excepto las piezas, que debemos pedirlas a Alexa mediante la frase anteriormente indicada para **crear una pieza**. La heurística es la misma que la utilizada en la simulación anterior. En caso de fallar la simulación se mostrará "*Heurística 0: GAME OVER*" y en

Figura 50: Simulación de TicTacToe o Tres en Raya

caso de completarla "*Heurística 36: COMPLETADO*". Para iniciarla tenemos que decir: "*Alexa, abre Objetos V. R. y comienza la simulación libre*".

- **Parar simulación**. Por último, indicamos una acción que nos permite detener la ejecución de la simulación actual y hace desaparecer todos los elementos propios de esta: **tablero, botones, cartel y cualquier pieza que pertenezca a la misma**. Para invocar esta acción deberemos decir: "*Alexa, abre Objetos V. R. y termina la simulación*".

Referencias

[1] Pillow Castle. *Superliminal*. 2020. URL: https://store.steampowered.com/app/1049410/Superliminal/.

[2] HackerPoet. *NonEuclidean*. 2018. URL: https://github.com/HackerPoet/NonEuclidean.

[3] Leap Motion. *Blocks*. 2016. URL: https://gallery.leapmotion.com/blocks/.

[4] Tech@Facebook. *From the lab to the living room: The story behind Facebook's Oculus Insight technology and a new era of consumer VR*. 2019. URL: https://tech.fb.com/the-story-behind-oculus-insight-technology/.

[5] OculusVR. *Introducing Oculus Quest 2, the Next Generation of All-in-One VR*. 2020. URL: https://developer.oculus.com/blog/introducing-oculus-quest-2-the-next-generation-of-all-in-one-vr/.

[6] Ben Lang. *Latest HTC Vives Are Shipping with Tweaked Base Stations, Redesigned Packaging*. 2017. URL: https://www.roadtovr.com/latest-vive-shipping-with-tweaked-base-stations-redesigned-packaging/.

[7] Kevin Viet Le. *Future Sport VR*. 2016. URL: https://www.hackster.io/kevinvle/future-sport-vr-034524.

[8] Ron Dagdag. *Control your "Earth Rover" in Virtual Reality*. 2016. URL: https://www.hackster.io/RONDAGDAG/control-your-earth-rover-in-virtual-reality-15a9fe.

[9] WebXR. *WebXR Device API*. URL: https://www.w3.org/TR/webxr/.

[10] Austin Winston. *Games SDK for Alexa*. 2018. URL: https://games-sdk-for-alexa.readthedocs.io/en/latest/index.html.

[11] Myron W. Krueger. *Artificial reality II*. Addison-Wesley, 1991. ISBN: 0201522608.

[12] Samuel Axon. *Unity at 10: For better—or worse—game development has never been easier*. Ars Technica, 2016. URL: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

[13] Fiona Kellhier. *Video game company grabs two buildings on Mission Street for big expansion*. San Francisco Business Times, 2018. URL: https://www.bizjournals.com/sanfrancisco/news/2018/08/24/video-game-unity-san-francisco-soma.html.

[14] Facebook Newsroom. *Facebook to acquire Oculus*. 2014. URL: https://about.fb.com/news/2014/03/facebook-to-acquire-oculus/.

[15] GSMArena. *FA 2014: Samsung Galaxy Note 4, Note Edge, Gear VR and Gear S hands-on*. 2015. URL: https://www.gsmarena.com/samsung_ifa_2014-review-1126.php.

[16] Nichole Marie Richardson. *One Giant Leap for Mankind*. Inc, 2013. URL: https://www.inc.com/30under30/nicole-marie-richardson/leap-motion-david-holz-michael-buckwald-2013.html.

[17] F. Weichert; D. Bachmann; B. Rudak; D. Fisseler. *Analysis of the Accuracy and Robustness of the Leap Motion Controller*. 2013. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3690061/.

[18] S. Vinoski. *Advanced Message Queuing Protocol*. IEEE Internet Computing, 2006. URL: https://ieeexplore.ieee.org/document/4012603.

[19] Amazon Alexa. *Alexa Skills Kit*. URL: https://developer.amazon.com/es-ES/alexa/alexa-skills-kit.

[20] Amazon Alexa. *Amazon Voice Service*. URL: https://developer.amazon.com/en-US/alexa/devices/alexa-built-in.

[21] Jeff Barr. *Amazon Lex – Build Conversational Voice Text Interfaces*. URL: https://aws.amazon.com/de/blogs/aws/amazon-lex-build-conversational-voice-text-interfaces/.

[22] Benjamin Black. *Benjamin Black – EC2 Origins*. 2009. URL: http://blog.b3k.us/2009/01/25/ec2-origins.html.

[23] Stuart Russell Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd ed.* Prentice Hall, 2010.

[24] LeapMotion. *Leap Motion Universal VR Dev Mount*. Thingiverse, 2016. URL: https://www.thingiverse.com/thing:1589212.

[25] Valem. *How to make a VR game in Unity - Part 3 - VR Shooter*. 2019. URL: https://www.youtube.com/watch?v=98gfkursxYI.

[26] Ultraleap. *Leap Motion Unity Modules*. URL: https://leapmotion.github.io/UnityModules/.

[27] Creepy Cat. *3D Scifi Kit Starter Kit*. 2019. URL: https://assetstore.unity.com/packages/3d/environments/3d-scifi-kit-starter-kit-92152.

[28] MesnikAnd. *Modular warehouse Free low-poly 3D model*. 2019. URL: https://www.cgtrader.com/free-3d-models/exterior/industrial/modular-warehouse.

[29] Cherokee21. *Oil barrel Free low-poly 3D model*. 2020. URL: https://www.cgtrader.com/free-3d-models/industrial/other/oil-barrel-0263285a-2df2-4587-a4f9-19da9e55f48a.

[30] Dmitrii Kutsenko. *PBR RPG/FPS Game Assets (Industrial Set v1.0)*. 2019. URL: https://assetstore.unity.com/packages/3d/environments/industrial/pbr-rpg-fps-game-assets-industrial-set-v1-0-146519.

[31] Queeboy. *Tools asset Free low-poly 3D model*. 2020. URL: https://www.cgtrader.com/free-3d-models/industrial/tool/unity-game-ready-tools-asset.

[32] OnkarShevkar. *Low Poly Buildings Free low-poly 3D model*. 2017. URL: https://www.cgtrader.com/free-3d-models/exterior/exterior-public/low-poly-buildings-8965cb09-8dae-4537-aeba-c28e90ebc1f8.

[33] CloudAMQP. *.NET Documentation*. URL: https://www.cloudamqp.com/docs/dotnet.html.

[34] CloudAMQP. *Javscript Documentation*. URL: https://www.cloudamqp.com/docs/nodejs.html.

[35] Amazon Alexa. *Standard Built-in Intents*. URL: https://developer.amazon.com/en-US/docs/alexa/custom-skills/standard-built-in-intents.html.

[36] Amazon Alexa. *Certification Requirements — Alexa Skills Kit*. URL: https://developer.amazon.com/en-US/docs/alexa/custom-skills/certification-requirements-for-custom-skills.html.

[37] Squaremo. *AMQP 0-9-1 library and client for Node.JS*. 2016. URL: https://github.com/squaremo/amqp.node/.

[38] Shekhar Gulati. *Building a REST API in Node.js with AWS Lambda, API Gateway, DynamoDB, and Serverless Framework*. 2009. URL: https://www.serverless.com/blog/node-rest-api-with-serverless-lambda-and-dynamodb.

[39] Serverless. *The Serverless Application Framework*. URL: https://www.serverless.com/.

[40] DDreaming Game. *Unity 3d Place Object On Grid*. URL: https://www.youtube.com/watch?v=9P5taAawy5I.

[41] Siccity. *GLTF Utility: Simple GLTF importer for Unity*. URL: https://github.com/Siccity/GLTFUtility/blob/master/README.md.

[42] KhronosGroup. *UnityGLTF*. URL: https://github.com/KhronosGroup/UnityGLTF.

[43] Sketchfab. URL: https://sketchfab.com.

[44] CGTrader. URL: https://www.cgtrader.com/.

[45] Free3D. URL: https://free3d.com/.

[46] Sketchfab. *Sketchfab for Developers*. URL: https://sketchfab.com/developers.

[47] Unity. *Optimizing your VR/AR Experiences*. URL: https://learn.unity.com/tutorial/optimizing-your-vr-ar-experiences#.